International Journal of Applied Metaheuristic Computing
Volume 14 • Issue 1

A Spark-Based Parallel Implementation of Arithmetic Optimization Algorithm

Maryam AlJame, Computer Engineering Department, College of Engineering and Petroleum, Kuwait University, Kuwait
https://orcid.org/0000-0003-3903-6839

Aisha Alnoori, Computer Engineering Department, College of Engineering and Petroleum, Kuwait University, Kuwait*
https://orcid.org/0000-0003-4276-7704

Mohammad G. Alfailakawi, Computer Engineering Department, College of Engineering and Petroleum, Kuwait University, Kuwait

Imtiaz Ahmad, Computer Engineering Department, College of Engineering and Petroleum, Kuwait University, Kuwait
https://orcid.org/0000-0002-0673-7324

ABSTRACT

Arithmetic optimization algorithm (AOA) is a recent population-based metaheuristic widely used for solving optimization problems. However, the emerging large-scale optimization problems pose a great challenge for AOA due to its prohibitive computational cost to traverse the huge solution space effectively. This article proposes a parallel Spark-AOA using Scala on Apache Spark computing platform. Spark-AOA leverages the intrinsic parallel nature of the population-based AOA and the native iterative in-memory computation support of Spark through resilient distributed datasets (RDD) to accelerate the optimization process. Spark-AOA divides the solutions population into several subpopulations that are distributed into multiple RDD partitions and manipulated concurrently. Simulation experiments on different benchmark functions with up to 1,000-dimension and three engineering design problems demonstrate that Spark-AOA outperforms considerably standard AOA and Spark-based implementations of two recent metaheuristics both in terms of run-time and solution quality.

KEYWORDS

AOA, Apache Spark, Arithmetic Optimization Algorithm, Cluster, Metaheuristic, Optimization, Parallel Computing, Parallelization

INTRODUCTION

The majority of real-world problems in many disciplines can be transformed into optimization problems and solved successfully by using optimization techniques. In recent years, nature-inspired population-based metaheuristics have been considered the state-of-the-art computational intelligence paradigms for solving complex optimization problems efficiently and effectively (Rahman et al.,

DOI: 10.4018/IJAMC.318642
*Corresponding Author

2021). The popularity and widespread success of metaheuristics is attributed to their robustness, adaptability, ease of implementation, and their ability to find optimal or near-optimal solutions in reasonable time by employing gradient-free search mechanisms on a population of solutions. A large number of metaheuristics have been developed in the literature and are classified based on the natural processes they mimic. The most commonly used classification for such metaheuristics are: evolution, swarm intelligence, physics, and human based (Meraihi et al., 2021). Classical metaheuristics include genetic algorithm (Holland, 1992), differential evolution (Storn & Price, 1997), and particle swarm optimization (Eberhart & Kennedy, 1995). More recent metaheuristics include slime mould (Li et al., 2020), sine cosine (Mirjalili, 2016), Harris hawk's (Heidari et al., 2019), and grey wolf optimizer (Mirjalili et al., 2014) among others (Hussain et al., 2019). There exist some other metaheursitics such as flower pollination optimization algorithm (Ding et al., 2021) and electromagnetism-like optimization algorithm (Chakraborty et al., 2022) which have been recently applied in biomedical image processing field.

However, no metaheuristic can efficiently solve all types of optimization problems according to the "No Free Lunch" theorem (Wolpert & Macready, 1997). Therefore, researchers have been continuously proposing new metaheuristics to deal with the ever-increasing complexity of real-world problems. Arithmetic Optimization Algorithm (AOA) is one of the newest metaheuristics proposed by Abualigah et al. (2021a) to solve various optimization problems. The AOA is inspired by the distribution behavior of four main arithmetic operators, namely, addition, subtraction, multiplication, and division to evolve solutions to achieve global optima. The AOA possesses some distinctive features such as simple and easy implementation, few tuning parameters, strong search ability that avoids falling into local minima, and considerably faster convergence rate by adaptively balancing exploration and exploitation phases. Despite being new, AOA has been very effective in solving real-world complex optimization problems in many fields as will be discussed in Section 2.

Large-scale optimization problems with high dimensionality are prevalent nowadays in diverse domains. The traditional serial implementation of metaheuristics does not scale well to solve such problems due to the high computational cost when evaluating population fitness and traversing large solution space (Abdelhafez et al., 2020). To address these challenges efficiently, there has been a growing interest in the parallelization of metaheuristics from both academia and industry (Coelho & Silva, 2021). These parallel metaheuristics leverage the intrinsic parallelism of population-based metaheuristics and the significant advancements in parallel computing devices such as multi-core CPUs/GPUs and distributed platforms such as Apache Hadoop and Spark to reduce execution time and improve solutions quality (Crainic, 2019; Hennessy & Patterson, 2017). Apache Spark, a distributed system framework, has recently experienced widespread usage due to its key features such as powerful API to easily parallelize application programs, support for iterative algorithms through in-memory computing, data distribution and processing on commodity clusters, and emerging cloud services with run-time load balancing, network performance, and fault-tolerance (Zaharia et al., 2016).

Therefore, many traditional metaheuristics such as genetic algorithm (Lu et al., 2020), particle swarm optimization (Al-Sawwa & Ludwig, 2020), differential evolution (He et al., 2021), whale optimization (AlJame et al., 2020), sine cosine (Alfailakawi et al., 2021), teaching-learning-based optimization (Wan et al., 2021), and grey wolf optimizer (Jarray et al., 2022a) have been successfully parallelized on Spark environments showing considerable performance gains for large scale problems. However, AOA being a recently proposed metaheuristic has not been parallelized under such an environment yet. Therefore, it is imperative to devise and investigate a distributed implementation of AOA to cope with large-scale problems efficiently. This work proposes Spark-AOA, a distributed implementation of AOA on Apache Spark to enhance its performance by reducing execution time as well as communication overhead. The main contributions of this study can be stated as the following:

- Devise a distributed Spark-AOA algorithm on Apache Spark environment with shuffle optimization to reduce communication overhead.

- Compare and analyze the performance of Spark-AOA to its serial version on benchmark functions as well as engineering design problems using various performance metrics such as execution time and solution quality.
- Compare and analyze Spark-AOA performance against recently parallelized metaheuristics on Spark environment such as whale optimization algorithm (AlJame et al., 2020) and sine cosine algorithm (Alfailakawi et al., 2021).

The remainder of this paper is organized as follows: Section 2 provides a brief description of AOA, related work, and a short overview of Apache Spark platform. The parallelization and implementation of AOA algorithm is described in Section 3. Section 4 discusses experimental results on benchmark test functions. The results of the proposed AOA in solving three engineering design problems as compared to serial version are given in Section 5. Finally, Section 6 concludes the paper and suggests future research directions.

BACKGROUND

AOA Overview

The AOA (Abualigah et al., 2021a) is a population-based metaheuristic optimization algorithm that utilizes basic arithmetic operators namely Division (D "$\div$"), Multiplication (M "$\times$"), Addition (A "$+$"), and Subtraction (S "$-$") to perform the optimization process to eventually reach optimal solution. Like other metaheuristic algorithms, the optimization process comprises of two main search phases: exploration and exploitation. The exploration phase generates diverse solutions to explore the search area while the exploitation phase focuses on searching local regions to reach the target solution.

The optimization process starts with a set of randomly generated candidate solutions to represent the initial population X as shown in Matrix (1):

$$X = \begin{bmatrix} x_{1,1} & \cdots & \cdots & x_{1,j} & x_{1,n-1} & x_{1,n} \\ x_{2,1} & \cdots & \cdots & x_{2,j} & \cdots & x_{2,n} \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ x_{N-1,1} & \cdots & \cdots & x_{N-1,j} & \cdots & x_{N-1,n} \\ x_{N,1} & \cdots & \cdots & x_{N,j} & x_{N,n-1} & x_{N,n} \end{bmatrix} \tag{1}$$

Where N denotes the number of randomly generated solutions in the initial population and n indicates the position of the solution.

In each iteration of AOA, the Math Optimizer Accelerated (MOA) function is computed to determine the search phase (i.e., Exploration or Exploitation). The MOA coefficient is calculated using Equation (2):

$$MOA\left(C\_Iter\right) = Min + C\_Iter \times \left(\frac{Max - Min}{M\_Iter}\right) \tag{2}$$

Where, $MOA\left(C\_Iter\right)$ represents the MOA function value during the current iteration; C_Iter and M_Iter denote current and maximum number of iterations, respectively; and Max and Min are the maximum and minimum values of the accelerated function, respectively.

Exploration Phase

The exploration phase of AOA algorithm is carried out by two arithmetic operators Division (D) and Multiplication (M) due to their ability to produce highly distributed values. The high dispersion characteristic of D and M aids the exploration phase to randomly generate a diverse range of candidate solutions to cover the search area. The exploration phase is constrained by the Math Optimizer Accelerated function for the case when $r1 > MOA$, where $r1$ is a random number. The position of the candidate solution is updated according to Equation (3) where the choice of D or M operators is constrained by the condition $r2 < 0.5$ ($r2$ is a random number):

$$x_{i,j}\left(C\_Iter+1\right)=\begin{cases} best\left(x_j\right)\div\left(MOP+\epsilon\right)\times\left(\left(UB_j-LB_j\right)\times\mu+LB_j\right), & r2<0.5 \\ best\left(x_j\right)\times MOP\times\left(\left(UB_j-LB_j\right)\times\mu+LB_j\right), & otherwise \end{cases} \tag{3}$$

Where $x_{i,j}\left(C\_Iter+1\right)$ represents the j th position of i th solution in the next iteration; $best\left(x_j\right)$ is the j th position in the best-obtained solution so far; ϵ is a small integer value; UB_j and LB_j represent the upper and lower bound of the j th position, respectively; and μ is a parameter used to control the exploration phase and is set to 0.5 (Abualigah et al., 2021a). The Math Optimizer Probability (MOP) coefficient is calculated using Equation (4):

$$MOP\left(C\_Iter\right)=1-\frac{C\_Iter^{1/\alpha}}{M\_Iter^{1/\alpha}} \tag{4}$$

Where $MOP\left(C\_Iter\right)$ represents MOP value at current iteration; C_Iter and M_Iter denote the current and maximum number of iterations, respectively; and α is a sensitivity parameter set to 5 (Abualigah et al., 2021a) and determines the efficiency of the exploitation over the iterations.

Exploitation Phase

The Subtraction (S) and Addition (A) operators are used to guide AOA's exploitation phase due to their low dispersion characteristic. This characteristic of S and A helps the exploitation phase to perform a thorough search in local regions of the search area to reach optimal solution with higher probability. The exploitation phase is controlled by the MOA function value for the case when $r1$ is not greater than MOA. The position of the candidate solution during the exploitation phase is updated according to Equation (5) where the choice of S or A operators is constrained by the condition $r3 < 0.5$:

$$x_{i,j}\left(C\_Iter+1\right)=\begin{cases} best\left(x_j\right)-MOP\times\left(\left(UB_j-LB_j\right)\times\mu+LB_j\right), & r3<0.5 \\ best\left(x_j\right)+MOP\times\left(\left(UB_j-LB_j\right)\times\mu+LB_j\right), & otherwise \end{cases} \tag{5}$$

To sum up, the AOA algorithm begins with a randomly generated set of candidate solutions that represents the initial population. As the algorithm iterates, solutions' positions are updated using D, M, S, and A operators with respect to the best-obtained solution so far. The MOA function value controls switching between exploration and exploitation phases to approach the target solution while maintaining solution diversity. The pseudo-code of AOA algorithm is given in Algorithm 1.

```
Algorithm 1 Pseudo-code for AOA algorithm

Initialize Arithmetic Optimization Algorithm parameters a, m.
Initialize the solutions' positions randomly. (Solutions: i= 1, ..., N.)
while (C_Iter < M_Iter) do
  Calculate Fitness Function (FF) for the given solutions
  Find best solution (Determined best so far).
  Update MOA and MOP values using Eq. (2) and Eq. (4), respectively.
  for (i= 1 to Solutions) do
    for (j= 1 to Positions) do
      Generate a random values between [0, 1] for (r1, r2, and r3)
      if r1 > MOA then
        Exploration phase
        if r2 > 0.5 then
          (1) Apply Division operator (D) and update
          the ith solutions' positions using Eq.(3).
        else
          (2) Apply Multiplication operator (M) and update
          the ith solutions' positions using Eq.(3).
        end if
      else
        Exploitation phase
        if r3 > 0.5 then
          (1) Apply Subtraction operator (S) and update
          the ith solutions' positions using Eq.(5).
        else
          (2) Apply Addition operator (A) and update
          the ith solutions' positions using Eq.(5).
        end if
      end if
    end for
  end for
  C_Iter= C_Iter+1
end while
Return the best solution (x).
```

LITERATURE REVIEW

AOA is a recent population-based metaheuristic proposed by Abualigah et al. (2021a), which has received wider acceptance due to its key advantages such as simplicity, few control parameters, and robust search competency. As of January 2023, AOA has been cited more than 915 times according to Google Scholar and has been applied successfully in various fields. Applications of AOA include thermoelectric power generation systems (Jarray et al., 2022b), damage assessment in functionally graded material plate structures (Khatir et al., 2021), optimized neural architecture search for early detection of Alzheimer's disease (Deepa & Chokkalingam, 2022), tuning hyper-parameters of bidirectional long short-term memory model for predicting the size of airborne particle bound metals (Almalawi et al., 2022), and localization problem in wireless sensor networks (Bhat & KV, 2022) to name a few.

Though the original AOA has shown considerable gains in performance against current metaheuristics, it may suffer from slow convergence, getting stuck in local optima, and inadequate exploitation. To further enhance AOA performance, researchers have proposed a variety of stochastic operators such as opposition-based learning (Abualigah et al., 2022; Yang et al., 2022), and Gaussian mutation mechanism and sinusoidal chaotic map (Xu et al., 2021) to address such issues. In another enhancement, such as in (Wang et al., 2021) AOA population is divided among multiple groups, where each group operates independently and exchanges information among randomly selected groups after fixed number of iterations. However, it was not implemented on a parallel computing system.

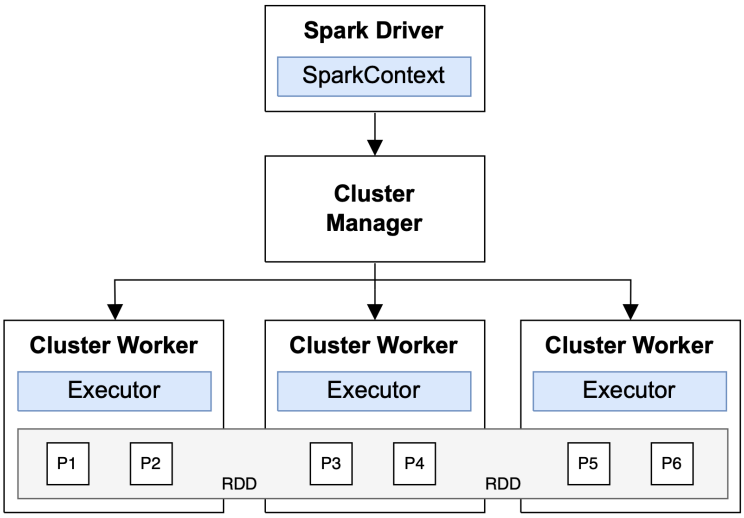
In addition, AOA performance has been enhanced by hybridizing it with other metaheuristics such as Aquila Optimizer (Abbassi et al., 2022; Zhang et al., 2022) to optimize voltaic cell parameters, Sine Cosine Algorithm (Abdel-Mawgoud et al., 2022) to determine the size and placement of battery energy storage devices, Electric Fish Optimization (Ibrahim et al., 2021) for feature selection problem, Salp Swarm Algorithm (Anjum et al., 2022) to optimize sizing and placement of generation units in radial distribution networks, Genetic Algorithm (Ewees et al., 2021) for feature selection problem, Golden Sine Algorithm (Liu et al., 2022) for solving industrial engineering problems, Differential Evolution (Abualigah et al., 2021b) for multilevel thresholding based image segmentation, and Slime Mould Algorithm (Zheng et al., 2021) to efficiently solve global optimization problems. Besides, several versions of AOA have been proposed by researchers to tackle specific optimization problems in different fields such as the binary versions for feature selection problem in machine learning field (Bansal et al., 2022; Dahou et al., 2022) and a discretized version to solve structural optimization in civil engineering field (Kaveh & Hamedani, 2022). Multi-objective versions of AOA based on non-dominance sorting have been proposed to solve real-world multi-objective optimization problems (Bahmanyar et al., 2022; Li et al., 2022; Premkumar et al., 2021). However, none of the reported works have considered the parallelization of AOA on distributed framework such as Apache Spark to speed-up computation as well as to further improve its search ability for large scale problems with higher dimensionality and complexity, which is the topic of this work.

Apache Spark

Spark is a powerful distributed computing framework built upon the core programming abstraction of Resilient Distributed Datasets (RDDs). RDDs represent immutable and fault-tolerant collection of data that can be distributed across multiple nodes in a cluster and can be manipulated in parallel. RDD can be created by loading a dataset from external storage systems such as Hadoop Distributed File System (HDFS) or Amazon S3. It can also be created by parallelizing a local collection in the program using SparkContext's parallelize() method. Spark supports a variety of programming languages, including Scala, R, Python, and Java and widely being used for a broad range of applications such as machine learning, data mining, and iterative algorithms (Zaharia et al., 2016).

Spark uses a master/slave architecture that consists of a driver node as a master and several worker nodes as slaves as shown in Figure 1. The Spark driver is responsible for converting user program into tasks and coordinating task scheduling on executors. The executors on worker nodes run the tasks and return the results to Spark driver. Additionally, they allocate in-memory storage for RDDs. The cluster manager allocates resources and launches executors (Zaharia et al., 2016). In fact, Spark can run on different cluster managers such as Hadoop YARN, Apache Mesos, and Spark's built-in standalone cluster manager. RDD has two types of operations: transformations and actions. Transformations are operations that produce a new RDD whereas actions compute results based on an RDD and either return results to driver program or save it to an external storage system. In essence, transformations return RDDs while actions return other data types. Transformations on RDDs are lazy evaluations which means that the transformation is not evaluated until the action is applied on the transformation. Lazy evaluation of transformations allows Spark to optimize the chain of operations before execution. Spark has two types of shared variables, namely, accumulators and broadcast variables. Accumulators

Figure 1. Apache spark architecture



combine values from worker nodes back to the driver program whereas broadcast variables enable the program to effectively distribute large values to all worker nodes.

THE PARALLELIZATION OF AOA ALGORITHM

The Arithmetic Optimization Algorithm is a population-based optimization algorithm that starts with randomly generated candidate solutions which diverge from near-optimal solution during exploration phase and converge toward near-optimal solution during exploitation phase over the course of iterations. This work proposes Spark-AOA, a Spark-based parallel implementation of AOA aiming to reduce execution time and communication overhead. The AOA algorithm's performance is affected by the increased computational complexity of fitness function evaluations that should be computed for each solution in the population. To enhance the performance, Spark-AOA splits the solutions population into subpopulations that are distributed into multiple partitions on different nodes in the cluster and manipulated concurrently. Furthermore, communication overhead is reduced by using broadcast variable to distribute best-obtained solution to all nodes. Moreover, the number of broadcasts is constrained by a user-defined parameter to further reduce communication overhead.

This section discusses the parallel design and implementation of the proposed Spark-AOA algorithm.

The Parallel Design

Distributed evolutionary algorithms (EAs) can be classified into two main categories based on how they divide computing tasks: population-distributed and dimension-distributed models. The population-distributed model involves distributing subpopulations among multiple processors or nodes, while the dimension-distributed model involves dividing the problem dimensions or subspaces among multiple processors or nodes. There are several subtypes of the population-distributed model, including master-slave, island, cellular, hierarchical, and pool models. The dimension-distributed model can be further divided into coevolution and multi-agent models (Gong et al., 2015).

The parallelization of AOA algorithm is based on the population-distributed model where subpopulations are distributed across multiple computing nodes in the cluster. This allows the model to be run more efficiently, as it can take advantage of the processing power of multiple devices

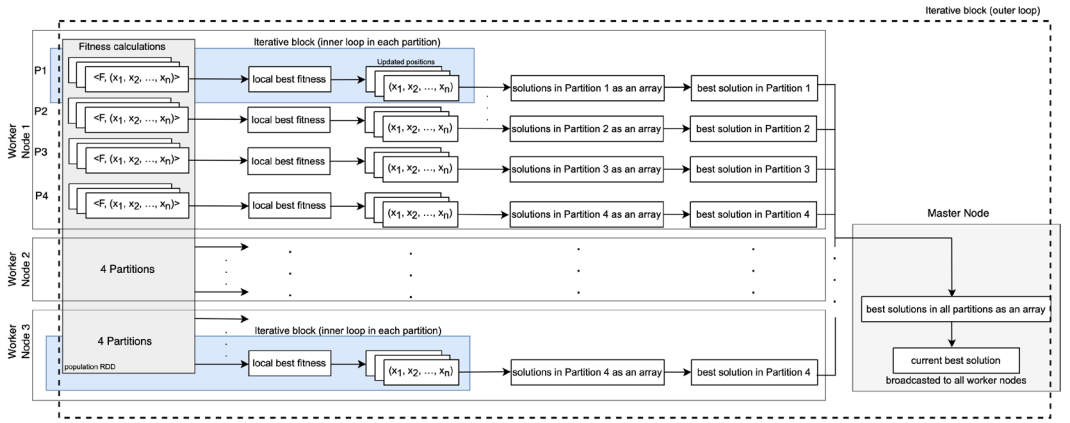
simultaneously. It can also be useful for handling large populations that may be computationally expensive to be processed by a single processor or node. To exchange information between subpopulations, the master-slave approach is used. In the master-slave model, the master is responsible for performing selection operations and broadcasting the best fitness, but it delegates the task of evaluating the fitness of subpopulations to slaves, as this constitutes the majority of the computing load. Because the evaluations of fitness in subpopulations are independent of each other, there is no need for communication between the slaves. This model is simple because communication only occurs between the master and the slaves, with the master sending subpopulations to the slaves and the slaves returning the corresponding fitness values to the master.

The parallelization of Spark-AOA optimization phase is illustrated in Figure 2. The population RDD is a data set of size N that is divided into P partitions and distributed to different worker nodes. Figure 2 shows three worker nodes where each node has four CPU cores and contains four partitions. Each subpopulation corresponds to a partition of the population RDD and each partition is assigned to one CPU core. All partitions work concurrently for several iterations to find the partition's best solution. The fitness function for each solution in the subpopulation is calculated simultaneously in all partitions. Then, the local best fitness in each partition is determined and all solutions in the subpopulations update their positions with respect to the local best fitness concurrently. Next, the best solution in each partition is obtained and the current best solution is selected as the global best fitness after a specific number of iterations.

Spark-AOA Algorithm

The pseudo-code of the proposed Spark-AOA is described in Algorithm 2. The initialization phase starts by initializing AOA parameters α and μ and sets the best fitness to infinity (line 1). Then, population X is initialized with N sets of randomly generated candidate solutions, each with n solution's positions that are constrained by upper bound and lower bound values (line 2). Spark parallelize() method is called from the driver program on master node to divide the population RDD into P partitions that are distributed across different nodes which can operate on the data in parallel. Then, Spark persist() method is called to store the subpopulations in memory of each node to reuse them in actions if needed hence avoid recomputation and allow faster future actions. In line 3, the fitness function is calculated for each solution in a subpopulation in each partition using mapPartitions() transformation. This results in an RDD [<F, (x_1, x_2, …, x_n)>] where each record is a key-value pair with fitness value F representing the key and solution (x_1, x_2, …, x_n) representing the value. The mapPartitions() is a powerful transformation that significantly reduces the amount of shuffling since

Figure 2. Parallelization of spark-AOA optimization phase



it operates on one partition at a time rather than on each RDD element. After that, the global best solution is determined by applying a map transformation on each partition on different nodes to find the best solution while a reduce action is applied to return the global best solution among all partitions (line 4). The glom() method is used to create an RDD that combines all elements within each partition into a list to reduce data shuffling across partitions. Lastly, the global best solution is broadcasted to all nodes in the cluster (line 5) and the solutions' positions are updated using Eq. (3) or Eq. (5) subject to the condition $r1 > MOA$ (line 6).

Following the initialization phase, Spark-AOA begins the optimization phase where the fittest solution in each subpopulation is identified at each iteration. When the main loop of the optimization phase starts, the mapPartitions() transformation calculates the fitness for each solution in the subpopulation in each partition (line 8). Next, the inner loop iterates M_Iter/m times so that each subpopulation will work on enhancing its own population iteratively (lines 9-20). In each iteration of the inner loop, the local best fitness in each partition is identified, then, MOA and MOP coefficients are calculated and random values ($r1$, $r2$, and $r3$) are generated according to the original AOA algorithm (lines 11-13). Subsequently, the solutions' positions are updated using Eq. (3) or Eq. (5) subject to the condition $r1 > MOA$ (lines 14-18) concluding one iteration of the inner loop. Eventually, all partitions stored in different worker nodes return the solutions as an array using the glom() method. The best solution in each partition is selected from the array and the current best solution among all partitions is identified (line 21). Following that, the fitness of the solution discovered in this iteration is compared to the global best fitness currently available. If a solution's fitness surpasses the current global best, the solution with the lowest fitness among all partitions is broadcasted (lines 22-25). The loop counter is then incremented (line 26) and the operation is continued until the predetermined number of iterations has been reached (line 27).

SPARK-AOA EVALUATION

This section evaluates the performance of the Spark-AOA as compared to the serial version of the algorithm in terms of exeution time and solution quality. Furthermore, Spark-AOA is compared with other recently parallelized metaheuristics in Spark environment namely Whale Optimization Algorithm (AlJame et al., 2020) with respect to execution time and Sine Cosine Algorithm (Alfailakawi et al., 2021) with respect to execution time and solution quality.

```
Algorithm 2 Pseudo-code of the Spark-AOA

Input: N = population size, P = number of partitions, n = dimension size,
Fn = objective function, M_Iter = maximum number of iterations,
m = inner loop iterations, C_Iter = 1
Output: best solution (fitness)
1: initialize AOA parameters a, m. initialize best fitness Fbest
to infinity
2: [(x1, x2, …, xn)] ← initialize solutions' positions randomly
and parallelize it across P partitions
3: [<F, (x1, x2, …, xn)>] ← calculate fitness for solutions' positions
4: <Fbest, (x1, x2, …, xn)> ← find best fitness
5: FagentBC ← broadcast best fitness <Fbest, (x1, x2, …, xn)>
6: [(x1, x2, …, xn)] ← update solutions' positions with respect
to Fbest
7: repeat
8:  [<F, (x1, x2, …, xn)>] ← calculate fitness for solutions' positions
9:  repeat  • inner loop in each partition
```

```
10:    find local best fitness (Flocal) in the partition
11:    update MOA value using Eq. (2)
12:    update MOP value using Eq. (4)
13:    generate random values between [0, 1] (r1, r2, and r3)
14:    if r1 > MOA then
15:       Exploration phase use Eq.(3) to update solutions'
position in the partition
16:    else
17:      Exploitation phase use Eq. (5) to update solutions' position in
the partition
18:    end if
19:    i = i + 1
20:  until i > M_Iter/m
21:  <FbestCurrent, (x1, x2, …, xn)> ← find current best fitness
among all partitions
22:  if FbestCurrent < FagentBC.value then
23:    FagentBC.destroy
24:    FagentBC ← broadcast best fitness <FbestCurrent, (x1, x2, …, xn)>
25:  end if
26:  C_Iter = C_Iter + 1
27: until C_Iter > m
28: return best fitness
```

Spark-AOA performance on nine benchmark functions is compared to its serial equivalent as well as Spark-SCA while focusing on dimensionality impact on execution time and solution quality. All three algorithms were tested using Amazon Elastic MapReduce (EMR), one of AWS's tools that offers a framework for big data processing on top of Amazon Elastic Compute Cloud (EC2). The used nodes were EC2 of type m4.xlarge with four vCPU and 16 Mem (GiB). Spark-AOA and Spark-SCA utilize one master node and three worker nodes while serial AOA has one master node. Equal numbers of function evaluations were employed in all implementations to allow for a fair comparison of Spark-AOA, serial AOA, and Spark-SCA. Population size for the serial version is set at 32, with 300 iterations. For Spark-AOA and Spark-SCA, a population size of 96 was used and a maximum number of iterations of 100 resulting in a total of 9,600 function evaluations for each implementation. Table 1 presents the speedup achieved by Spark-AOA as compared to the serial version as well as Spark-SCA. The average values of best fitness, standard deviation (STD), best and worst fitness values achieved for all algorithms across 30 separate runs for each dimension size are reported in Table 2.

Table 1 and 2 show the results for three different benchmark dimensions, in particular 50, 250, and 1,000. The nine benchmark functions used in these experiments include three unimodal functions, namely Sphere (F1), Schwefel 2.21 (F4), and Rosenbrock (F5), three multimodal functions, namely Rastrigin (F9), Ackley (F10), and Griewank (F11), and three composite functions, namely CF3 (F16), CF4 (F17), and CF5 (F18). Benchmark functions details are described in the appendix. The main purpose of this experiment was to investigate algorithm performance with respect to speedup and solution quality as a function of problem size. The speedup is computed as the execution time of algorithm x divided by the execution time of Spark-AOA algorithm, where x is either serial AOA or Spark-SCA.

As shown in Table 1, Spark-AOA performed poorly as compared to the serial version and Spark-SCA for all benchmarks when using dimensions 50 except for F5. With dimension of 250, Spark-AOA resulted in a speedup for all multimodal functions and one unimodal function (F5) when compared to the serial version. On the other hand, as compared to Spark-SCA, Spark-AOA provided better

performance for all benchmarks with dimenison 250 except for F4, F9, and F17. For dimension 1000, Spark-AOA is the best performing algorithm as compared to the remaining ones for all benchmarks. These results suggest that Spark-AOA is not appropriate when solving small or medium-sized problems, in particular problems charaterised by having a single minima such as unimodal functions. However, Spark-AOA can achieve significant speedup when addressing higher-dimensional problems.

The results presented in Table 2 indicate that Spark-AOA algorithm performs well on all benchmark functions in terms of solution quality regardless of problem dimension. A comparison with the serial AOA shows that Spark-AOA obtained the best fitness for all benchmark functions, except for F16 with dimension 50. Furthermore, it can be observed that the proposed algorithm consistently outperforms the Spark-SCA algorithm in terms of solution quality, except for F5 with dimension 1000. In summary, the results suggest that Spark-AOA exhibits superior solution quality when compared to both the sequential version and Spark-SCA algorithm.

The second part of the experiment compared the average running time of the proposed parallel AOA algorithm to the Spark-based Whale Optimization Algorithm (Spark-WOA). The Spark-WOA (AlJame et al., 2020) implementation was tested on an EMR Yarn cluster comprising ten Amazon EC2 instances, including one master node and nine worker nodes. These instances were of the general purpose m4.large type, each with 2 vCPUs and 8 GiB of memory. The cluster had a total of 20 cores. Alternatively, Spark-AOA was tested on the same cluster configuration explained earlier with one master node and three worker nodes. The used nodes were EC2 of type m4.xlarge with four vCPU and 16 Mem (GiB). Therefore, the total number of cores in the cluster is 16 cores. The performance of Spark-AOA was compared to Spark-WOA using five different benchmarks (F1, F4, F5, F9, and F10). To ensure a fair comparison, Spark-AOA was executed with a population size of 512 and a maximum number of iterations of 70, resulting in a same number of function evaluations as Spark-WOA. The benchmark dimension was set to 1,000. Additionally, Spark-AOA was run 30 times for each benchmark, and average execution times for these runs are reported. Table 3 shows average execution time for Spark-AOA and Spark-WOA. It is apparent from the table that Spark-AOA is approximately 40 times faster than Spark-WOA on average for all benchmark functions. The significant improvement in Spark-AOA's speed could be attributed to several factors such as reducing communication overhead by limiting the number of broadcasts to worker nodes and the nature of the two metaheuristics. Therefore, it can be concluded that Spark-AOA algorithm has superior computational performance compared to the Spark-WOA algorithm.

ENGINEERING DESIGN PROBLEMS

This section compares Spark-AOA to its serial counterpart on real optimization problems such as constrained engineering design problems namely welded beam (Ragsdell & Phillips, 1976), tension/compression spring (Arora, 2004; Belegundu & Arora, 1985), and pressure vessel (Kannan & Kramer, 1994; Moss, 2004). The main goal of applying optimization to engineering problems is to reduce the values of design parameters and consequently overall design cost.

The objective of the welded beam design problem is to minimize manufacturing cost by obtaining the best value of four optimization variables, namely, thickness of weld (h), length of attached part of beam (l), beam thickness (b), and height of the bar (t) while using shear stress (τ), bending stress (θ), buckling load (P), and deflection (δ) as the constraints. The mathematical representation of the welded beam design problem, shown in Figure 3, is described as follows:Consider

$$\vec{x} = \begin{bmatrix} x_1 & x_2 & x_3 & x_4 \end{bmatrix} = \begin{bmatrix} h & l & t & b \end{bmatrix}$$

Table 1. Spark-AOA speedup vs serial-AOA and spark-SCA

Function	Dimension	Speedup	
		Spark-AOA vs Serial-AOA	Spark-AOA vs Spark-SCA
F1	50	0.211	0.822
	250	0.968	**1.098**
	1,000	**2.885**	**2.325**
F4	50	0.219	0.933
	250	0.763	0.937
	1,000	**2.734**	**2.061**
F5	50	0.345	**1.008**
	250	**1.302**	**1.440**
	1,000	**3.033**	**2.066**
F9	50	0.308	0.943
	250	**1.142**	0.884
	1,000	**3.119**	**1.641**
F10	50	0.341	**2.402**
	250	**1.238**	**1.413**
	1,000	**3.379**	**1.554**
F11	50	0.402	0.882
	250	**1.598**	**1.547**
	1,000	**3.714**	**1.771**
F16	50	0.198	0.903
	250	0.843	**1.201**
	1,000	**2.548**	**1.852**
F17	50	0.166	0.861
	250	0.303	0.393
	1,000	**2.682**	**1.875**
F18	50	0.172	0.952
	250	0.784	**1.291**
	1,000	**2.619**	**1.642**

Minimize

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

Subject to

$$g_1\left(\vec{x}\right) = \tau\left(\vec{x}\right) - 13,600 \leq 0$$

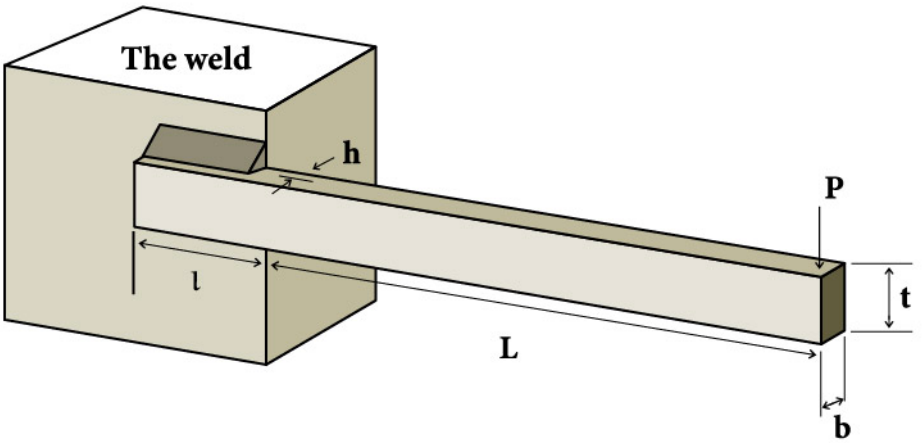
Table 2. Normalized fitness values

F		Spark-SCA			Spark-AOA			Serial-AOA		
	Dim	50	250	1,000	50	250	1,000	50	250	1,000
F1	Average	2.13E-04	3.92E-04	9.87E-04	**7.61E-210**	**8.34E-205**	**0.00E+00**	9.45E-03	2.13E-01	1.00E+00
	STD	2.85E-02	5.17E-02	1.23E-01	0.00E+00	0.00E+00	0.00E+00	1.76E-01	1.00E+00	3.76E-01
	Best	0.00E+00	0.00E+00	0.00E+00	1.07E-230	4.33E-231	1.17E-231	9.61E-04	1.10E-01	1.00E+00
	Worst	6.09E-03	1.11E-02	2.64E-02	2.62E-208	2.43E-203	0.00E+00	2.79E-02	2.39E-01	1.00E+00
F4	Average	7.10E-03	1.63E-03	1.83E-02	**9.23E-105**	**0.00E+00**	**6.86E-104**	9.14E-01	9.86E-01	1.00E+00
	STD	4.13E-01	6.37E-02	9.43E-01	4.34E-103	0.00E+00	5.79E-102	1.00E+00	7.33E-02	1.82E-02
	Best	1.25E-08	3.22E-12	2.26E-08	2.09E-115	4.06E-116	0.00E+00	5.95E-01	9.76E-01	1.00E+00
	Worst	1.41E-01	1.88E-02	3.12E-01	1.47E-103	0.00E+00	2.03E-102	9.59E-01	9.91E-01	1.00E+00
F5	Average	1.65E-07	1.69E-03	**0.00E+00**	**3.27E-09**	**1.67E-08**	6.69E-08	2.77E-02	2.24E-01	1.00E+00
	STD	2.33E-05	3.82E-01	3.15E-09	0.00E+00	4.66E-11	1.63E-10	3.31E-01	5.19E-01	1.00E+00
	Best	3.53E-09	1.79E-08	0.00E+00	3.53E-09	1.79E-08	7.20E-08	9.99E-03	2.05E-01	1.00E+00
	Worst	2.13E-06	4.82E-02	0.00E+00	2.77E-09	1.57E-08	6.42E-08	4.01E-02	2.35E-01	1.00E+00
F9	Average	4.97E-03	2.27E-02	2.37E-01	**0.00E+00**	**0.00E+00**	**0.00E+00**	1.44E-02	1.91E-01	1.00E+00
	STD	1.86E-02	1.32E-01	1.00E+00	0.00E+00	0.00E+00	0.00E+00	9.26E-03	3.60E-02	1.01E-01
	Best	0.00E+00	0.00E+00	4.16E-14	0.00E+00	0.00E+00	0.00E+00	7.41E-03	1.74E-01	1.00E+00
	Worst	1.66E-02	1.78E-01	1.00E+00	0.00E+00	0.00E+00	0.00E+00	1.29E-02	1.23E-01	5.68E-01
F10	Average	2.19E-02	1.73E-02	1.33E-02	**0.00E+00**	**0.00E+00**	**0.00E+00**	7.58E-01	9.79E-01	1.00E+00
	STD	4.71E-01	2.86E-01	3.71E-01	0.00E+00	0.00E+00	0.00E+00	1.00E+00	8.96E-02	6.94E-02
	Best	0.00E+00	4.02E-11	9.73E-10	2.35E-17	2.35E-17	2.35E-17	4.03E-01	9.90E-01	1.00E+00
	Worst	3.29E-01	2.14E-01	3.09E-01	0.00E+00	0.00E+00	0.00E+00	9.59E-01	9.97E-01	1.00E+00
F11	Average	9.80E-04	1.56E-03	4.31E-02	**0.00E+00**	**0.00E+00**	**0.00E+00**	7.68E-03	1.80E-01	1.00E+00
	STD	2.38E-02	3.80E-02	1.00E+00	0.00E+00	0.00E+00	0.00E+00	3.18E-02	1.39E-01	5.73E-01
	Best	0.00E+00	0.00E+00	0.00E+00	0.00E+00	0.00E+00	0.00E+00	9.21E-05	1.70E-01	1.00E+00
	Worst	2.333E-02	3.73E-02	9.83E-01	0.00E+00	0.00E+00	0.00E+00	2.10E-02	1.98E-01	1.00E+00
F16	Average	1.00E+00	6.36E-01	6.12E-01	6.70E-01	**0.00E+00**	**4.87E-01**	**2.60E-01**	8.58E-02	5.37E-01
	STD	1.00E+00	6.38E-01	4.42E-01	8.70E-01	0.00E+00	8.94E-01	6.70E-01	6.57E-01	8.13E-01
	Best	0.00E+00	3.64E-01	1.00E+00	8.08E-02	0.00E+00	2.02E-02	0.00E+00	0.00E+00	0.00E+00
	Worst	1.00E+00	5.36E-01	8.41E-01	3.78E-01	0.00E+00	3.79E-01	3.31E-01	3.49E-01	3.72E-01
F17	Average	1.53E-01	2.24E-01	2.27E-01	**1.02E-02**	**7.43E-02**	**0.00E+00**	1.00E+00	6.79E-01	8.55E-01
	STD	1.15E-01	2.07E-01	1.68E-01	2.85E-02	1.12E-01	0.00E+00	9.38E-01	8.21E-01	1.00E+00
	Best	3.84E-03	6.14E-01	2.51E-01	0.00E+00	0.00E+00	7.68E-03	1.00E+00	1.50E-01	0.00E+00
	Worst	3.49E-02	2.06E-01	1.01E-01	2.33E-03	7.89E-02	0.00E+00	7.47E-01	1.00E+00	8.75E-01
F18	Average	4.79E-01	6.33E-01	5.27E-01	**1.07E-01**	**1.55E-01**	**0.00E+00**	6.86E-01	1.00E+00	4.78E-01
	STD	8.54E-01	8.26E-01	6.71E-01	1.55E-01	2.81E-01	0.00E+00	1.00E+00	9.46E-01	5.71E-01
	Best	1.36E-01	1.00E+00	9.23E-01	1.44E-02	9.06E-02	5.13E-02	0.00E+00	0.00E+00	4.01E-04
	Worst	1.00E+00	6.39E-01	6.19E-01	1.78E-01	3.99E-01	0.00E+00	6.35E-01	5.90E-01	4.56E-01

Table 3. Spark-AOA and spark-WOA execution time comparison

Benchmark	Execution time (seconds)	
	Spark-AOA	Spark-WOA
F1	1.02	52.10
F4	1.05	51.96
F5	1.58	48.98
F9	1.17	49.35
F10	1.27	48.61

Figure 3. Welded beam design problem



$$g_2\left(\vec{x}\right)=\sigma\left(\vec{x}\right)-30,000\leq 0$$

$$g_3\left(\vec{x}\right)=x_1-x_4\leq 0$$

$$g_4\left(\vec{x}\right)=0.10471x_1^2+0.04811x_4x_3\left(14+x_2\right)-5.0\leq 0$$

$$g_5\left(\vec{x}\right)=0.125-x_1\leq 0$$

$$g_6\left(\vec{x}\right)=\delta\left(\vec{x}\right)-0.25\leq 0$$

$$g_7\left(\vec{x}\right)=6,000-P\left(\vec{x}\right)\leq 0$$

Where

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

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

$$\tau'' = \frac{MR}{J}$$

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

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

$$J = 2\left\{\sqrt{2}\, x_1 x_2 \left[\frac{x_2^2}{4} + \left(\frac{x_1 + x_3}{2}\right)^2\right]\right\}$$

$$\sigma\left(\vec{x}\right) = \frac{504{,}000}{x_4 x_3^2}$$

$$\delta\left(\vec{x}\right) = \frac{65{,}856{,}000}{\left(30\times10^6\right) x_4 x_3^3}$$

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

The intervals for the optimization variables are as follows:

$$0.10 \le h, b \le 2.0$$
$$0.10 \le l, t \le 10.0$$

The second design problem is the tension/compression spring design problem that aims to minimize the weight of the tension/compression spring to satisfy the design constraints. Three design variables need to be considered in this problem namely, mean coil diameter (D), number of spring's active coil (N), and diameter of the wire (d). The tension/compression spring design problem schematic is shown in Figure 4 and the mathematical formulation of the design problem is defined as follows:Consider

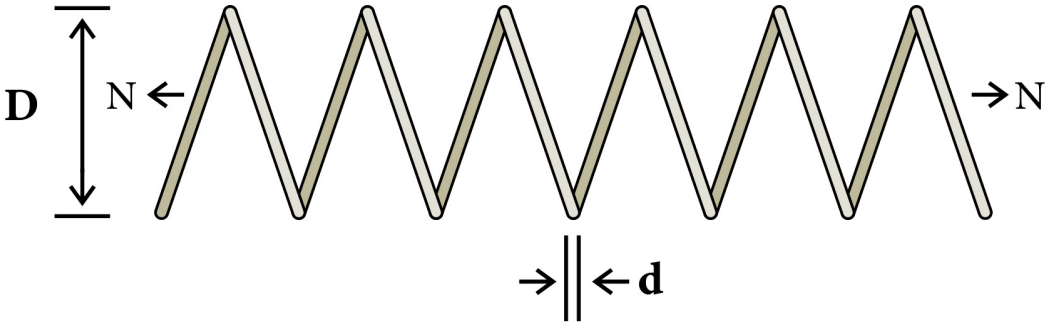
$$\vec{x} = \begin{bmatrix} x_1 & x_2 & x_3 \end{bmatrix} = \begin{bmatrix} d & D & N \end{bmatrix}$$

Minimize

$$f\left(\vec{x}\right) = x_1^2 x_2 x_3 + 2x_1^2 x_2$$

Subject to

Figure 4. Tension/compression spring design problem



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

$$g_2\left(\vec{x}\right) = \frac{4x_2^2 - x_1 x_2}{12566\left(x_2 x_1^3 - x_1^4\right)} + \frac{1}{5108 x_1^2} - 1 \leq 0$$

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

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

The intervals for the optimization variables are as follows:

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

The pressure vessel design problem schematic is shown in Figure 5. The pressure vessel consists of a cylindrical part that is capped with hemispherical heads at both ends. Four design parameters need to be optimized to minimize the total cost, namely thickness of pressure vessel (T_s), head thickness (T_h), length of the cylindrical part excluding the heads (L), and the inner radius (R). The mathematical representation of pressure vessel design problem is described as follows:Consider

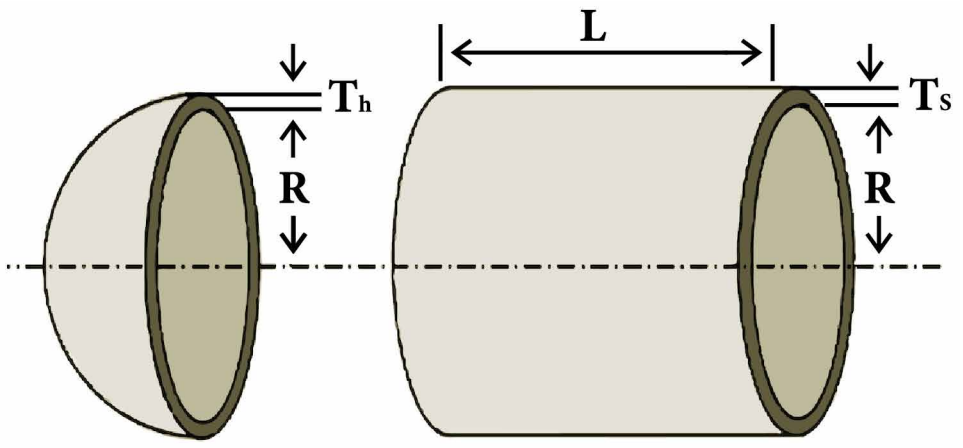
$$\vec{x} = \left[x_1 \quad x_2 \quad x_3 \quad x_4\right] = \left[T_s \quad T_h \quad R \quad L\right]$$

Minimize

$$f\left(\vec{x}\right) = 0.6224 x_1 x_3 x_4 + 1.7781 x_2 x_3^2 + 3.1661 x_1^2 x_4 + 19.84 x_1^2 x_3$$

Subject to

Figure 5. Pressure vessel design problem



$$g_1\left(\vec{x}\right) = -x_1 + 0.0193x_3 \leq 0$$

$$g_2\left(\vec{x}\right) = -x_2 + 0.00954x_3 \leq 0$$

$$g_3\left(\vec{x}\right) = -\pi x_3^2 x_4 - \frac{4}{3}\pi x_3^3 + 1296000 \leq 0$$

$$g_4\left(\vec{x}\right) = x_4 - 240 \leq 0$$

The intervals for the optimization variables are as follows:

$$0.0625 \leq T_s, T_h \leq 6.1875$$

$$10 \leq R, L \leq 200$$

The proposed algorithm (Spark-AOA) and its serial version were applied for solving the welded beam, tension/compression spring, and pressure vessel design problems and their performance in term of execution time, speedup, and overall cost were compared. The results are given in Tables 4, 5, and 6. All tables include maximum number of iterations and population size (number of agents). The number of function evaluations and cluster settings were the same for both algorithms, similar to what was done in the previous section. The reported results in this section are the average of 30 independent runs. For all three engineering design problems, the proposed Spark-AOA algorithm outperforms the serial version in terms of execution time. In addition, the proposed algorithm offers at least a two-fold speedup over the serial version when the population size exceeds 8000. Evidently, the speedup grows as population size increases in most cases. In terms of cost, Spark-AOA performs relatively poorly compared to the serial version in most cases. Both implementations resulted in a comparable cost for the tension/compression spring design problem. As for the welded beam design problem, Spark-AOA gives a higher cost for four out of five cases with an average increase of 5% as compared to the serial version. Furthermore, Spark-AOA resulted in a higher cost for the pressure vessel design problem as compared to the serial version. Although the cost is typically higher, Spark-AOA is more efficient in terms of run-time, providing at least two times the speedup over the serial implementation.

Table 4. Welded beam design problem results

Algorithm	# of evaluations		Optimization variables				Cost	Time (sec)	Speedup
	Max. iter.	Agents	h	l	t	b			
AOA	250	500	0.204	4.344	9.491	0.238	2.133	0.606	1.603
Spark-AOA	200	625	0.177	5.865	9.217	0.235	2.241	0.378	
AOA	500	2,000	0.179	5.546	9.674	0.218	2.127	3.907	3.108
Spark-AOA	125	8,000	0.189	5.684	9.024	0.243	2.272	1.257	
AOA	600	4,000	0.184	5.332	9.752	0.209	2.080	8.748	3.22
Spark-AOA	100	24,000	0.198	4.983	8.963	0.242	2.167	2.717	
AOA	300	10,000	0.186	4.859	9.743	0.216	2.058	10.745	2.976
Spark-AOA	50	60,000	0.197	4.591	8.927	0.234	2.044	3.611	
AOA	120	75,000	0.179	4.590	9.564	0.212	1.964	29.357	3.023
Spark-AOA	100	90,000	0.203	4.426	8.978	0.235	2.049	9.712	

Table 5. Tension/compression spring design problem results

Algorithm	# of evaluations		Optimization variables			Cost	Time (sec)	Speedup
	Max. iter.	Agents	d	D	N			
AOA	250	500	0.061	0.673	9.526	0.019	0.434	1.219
Spark-AOA	200	625	0.055	0.462	12.627	0.018	0.356	
AOA	500	2,000	0.056	0.518	11.726	0.017	2.796	2.461
Spark-AOA	125	8,000	0.053	0.387	13.259	0.016	1.136	
AOA	600	4,000	0.055	0.456	12.173	0.015	6.463	2.608
Spark-AOA	100	24,000	0.055	0.447	10.522	0.015	2.478	
AOA	300	10,000	0.052	0.374	13.233	0.014	8.085	2.586
Spark-AOA	50	60,000	0.055	0.428	10.368	0.015	3.126	
AOA	120	75,000	0.054	0.399	11.353	0.014	24.628	2.927
Spark-AOA	100	90,000	0.054	0.413	10.53	0.014	8.414	

CONCLUSION

Nature-inspired population-based metaheuristics including the recently proposed AOA are state-of-the-art computational intelligence paradigms for exploring efficiently the search space of complex optimization problems. This article developed and implemented a distributed version of AOA called Spark-AOA on Apache Spark environment in order to accelerate the optimization process. To our knowledge, AOA has previously not been implemented on the Spark platform. Spark-AOA harnessed the in-memory parallel computing capabilities of Spark and the intrinsic parallelism in population-based metaheuristics to reduce computational time. Spark-AOA divided the population into several subpopulations and each subpopulation evolved concurrently on a different node in the cluster. The

Table 6. Pressure vessel design problem results

Algorithm	# of evaluations		Optimization variables				Cost	Time (sec)	Speedup
	Max. iter.	Agents	T_s	T_h	R	L			
AOA	250	500	1.100	0.688	49.362	137.591	8419.168	0.447	1.211
Spark-AOA	200	625	1.128	0.844	47.001	164.798	10146.855	0.369	
AOA	500	2,000	1.015	0.620	46.949	150.831	7727.2456	2.934	2.359
Spark-AOA	125	8,000	1.048	0.622	48.254	145.303	8558.302	1.244	
AOA	600	4,000	0.938	0.594	46.954	148.626	7141.491	6.798	2.552
Spark-AOA	100	24,000	1.041	0.663	48.166	140.632	8507.578	2.664	
AOA	300	10,000	0.911	0.574	44.284	166.026	7079.079	8.648	2.501
Spark-AOA	50	60,000	1.001	0.619	47.985	137.092	7864.746	3.458	
AOA	120	75,000	0.883	0.541	43.311	173.038	6892.649	26.115	2.734
Spark-AOA	100	90,000	0.989	0.594	47.414	140.684	7630.083	9.552	

communication between subpopulations was carried out by using a broadcast variable to distribute best-obtained solution to all nodes at specific iteration intervals. To evaluate the performance of Spark-AOA, various simulation experiments were conducted on different benchmarks and engineering design problems on Amazon Web Services (AWS) public cloud. The important conclusions of this article can be summarized as follows:

- Spark-AOA improved the speedup and solution quality to solve large scale optimization problems due to the exploitation of population level parallelism as compared with standard AOA.
- Experimental results revealed that the high-dimensional optimization problems requiring a large number of time-consuming fitness evaluations per iteration benefit most from parallelization. On the contrary, low-dimensional problems requiring a small number of short fitness evaluations per iteration showed limited scalability.
- Spark-AOA demonstrated higher speedup and better solution quality as compared with Spark-based implementations of recent metaheuristics such as Whale Optimization Algorithm and Sine Cosine Algorithm.

A possible limitation of the proposed approach is that it did not exploit the parallelism at the dimension level of the optimization problem to further enhance the speedup. In addition, computation to communication cost trade-off among subpopulations needs further investigation. Based on these observations of the study, the authors plan to pursue the following research directions in order to improve the versatility and robustness of our approach.

- Study the effects of more complex inter-subpopulations communication and exploitation of parallelism at the dimension level for each individual of subpopulation to harness benefits of Spark-AOA on compute-intensive real-life optimization problems.
- Investigate the parallelization of hybrid versions of AOA including multi-objective AOA in Spark environment.
- Explore opportunities of parallelizing AOA with GPU to enhance its competitive effectiveness and efficiency with respect to the solution quality and the execution time.

ACKNOWLEDGMENT

Competing Interests

The authors declare that they have no conflict of interest.

Funding Agency

There is no funding associated with this research.

REFERENCES

Abbassi, A., Mehrez, R. B., Touaiti, B., Abualigah, L., & Touti, E. (2022). Parameterization of photovoltaic solar cell double-diode model based on improved arithmetic optimization algorithm. *Optik (Stuttgart)*, *253*, 168600. doi:10.1016/j.ijleo.2022.168600

Abdel-Mawgoud, H., Fathy, A., & Kamel, S. (2022). An effective hybrid approach based on arithmetic optimization algorithm and sine cosine algorithm for integrating battery energy storage system into distribution networks. *Journal of Energy Storage*, *49*, 104154. doi:10.1016/j.est.2022.104154

Abdelhafez, A., Luque, G., & Alba, E. (2020). Parallel execution combinatorics with metaheuristics: Comparative study. *Swarm and Evolutionary Computation*, *55*, 100692. doi:10.1016/j.swevo.2020.100692

Abualigah, L., Almotairi, K. H., Al-qaness, M. A., Ewees, A. A., Yousri, D., Abd Elaziz, M., & Nadimi-Shahraki, M. H. (2022). Efficient text document clustering approach using multi-search Arithmetic Optimization Algorithm. *Knowledge-Based Systems*, *248*, 108833. doi:10.1016/j.knosys.2022.108833

Abualigah, L., Diabat, A., Mirjalili, S., Abd Elaziz, M., & Gandomi, A. H. (2021a). The arithmetic optimization algorithm. *Computer Methods in Applied Mechanics and Engineering*, *376*, 113609. doi:10.1016/j.cma.2020.113609

Abualigah, L., Diabat, A., Sumari, P., & Gandomi, A. H. (2021b). A novel evolutionary arithmetic optimization algorithm for multilevel thresholding segmentation of covid-19 ct images. *Processes (Basel, Switzerland)*, *9*(7), 1155. doi:10.3390/pr9071155

Al-Sawwa, J., & Ludwig, S. A. (2020). Parallel particle swarm optimization classification algorithm variant implemented with Apache Spark. *Concurrency and Computation*, *32*(2), e5451. doi:10.1002/cpe.5451

Alfailakawi, M. G., Aljame, M., & Ahmad, I. (2021). Parallel and distributed implementation of sine cosine algorithm on apache spark platform. *IEEE Access: Practical Innovations, Open Solutions*, *9*, 77188–77202. doi:10.1109/ACCESS.2021.3082026

AlJame, M., Ahmad, I., & Alfailakawi, M. (2020). Apache spark implementation of whale optimization algorithm. *Cluster Computing*, *23*(3), 2021–2034. doi:10.1007/s10586-020-03162-7

Almalawi, A., Khan, A. I., Alsolami, F., Alkhathlan, A., Fahad, A., Irshad, K., Alfakeeh, A. S., & Qaiyum, S. (2022). Arithmetic optimization algorithm with deep learning enabled airborne particle-bound metals size prediction model. *Chemosphere*, *303*, 134960. doi:10.1016/j.chemosphere.2022.134960 PMID:35580643

Anjum, Z. M., Said, D. M., Hassan, M. Y., Leghari, Z. H., & Sahar, G. (2022). Parallel operated hybrid Arithmetic-Salp swarm optimizer for optimal allocation of multiple distributed generation units in distribution networks. *PLoS One*, *17*(4), e0264958. doi:10.1371/journal.pone.0264958 PMID:35417475

Arora, J. (2004). *Introduction to optimum design*. Elsevier. doi:10.1016/B978-012064155-0/50012-4

Bahmanyar, D., Razmjooy, N., & Mirjalili, S. (2022). Multi-objective scheduling of IoT-enabled smart homes for energy management based on Arithmetic Optimization Algorithm: A Node-RED and NodeMCU module-based technique. *Knowledge-Based Systems*, *247*, 108762. doi:10.1016/j.knosys.2022.108762

Bansal, P., Gehlot, K., Singhal, A., & Gupta, A. (2022). Automatic detection of osteosarcoma based on integrated features and feature selection using binary arithmetic optimization algorithm. *Multimedia Tools and Applications*, *81*(6), 8807–8834. doi:10.1007/s11042-022-11949-6 PMID:35153620

Belegundu, A. D., & Arora, J. S. (1985). A study of mathematical programming methods for structural optimization. Part I: Theory. *International Journal for Numerical Methods in Engineering*, *21*(9), 1583–1599. doi:10.1002/nme.1620210904

Bhat, S. J., & K v, S. (2022). A localization and deployment model for wireless sensor networks using arithmetic optimization algorithm. *Peer-to-Peer Networking and Applications*, *15*(3), 1473–1485. doi:10.1007/s12083-022-01302-x

Chakraborty, S., & Mali, K. (2022). SUFEMO: A superpixel based fuzzy image segmentation method for COVID-19 radiological image elucidation. *Applied Soft Computing*, *129*, 109625. doi:10.1016/j.asoc.2022.109625 PMID:36124000

Coelho, P., & Silva, C. (2021). Parallel Metaheuristics for shop scheduling: Enabling industry 4.0. *Procedia Computer Science*, *180*, 778–786. doi:10.1016/j.procs.2021.01.328

Crainic, T. (2019). Parallel metaheuristics and cooperative search. Michel Gendreau, Jean-Yves Potvin. In *Handbook of metaheuristics* (pp. 419–451). Springer. doi:10.1007/978-3-319-91086-4_13

Dahou, A., Al-qaness, M. A., Abd Elaziz, M., & Helmi, A. (2022). Human activity recognition in IOHT applications using arithmetic optimization algorithm and deep learning. *Measurement*, *199*, 111445. doi:10.1016/j.measurement.2022.111445

Deepa, N., & Chokkalingam, S. P. (2022). Optimization of VGG16 utilizing the Arithmetic Optimization Algorithm for early detection of Alzheimer's disease. *Biomedical Signal Processing and Control*, *74*, 103455. doi:10.1016/j.bspc.2021.103455

Ding, W., Chakraborty, S., Mali, K., Chatterjee, S., Nayak, J., Das, A. K., & Banerjee, S. (2021). An unsupervised fuzzy clustering approach for early screening of COVID-19 from radiological images. *IEEE Transactions on Fuzzy Systems*, *30*(8), 2902–2914. doi:10.1109/TFUZZ.2021.3097806 PMID:36345371

Eberhart, R., & Kennedy, J. (1995, October). A new optimizer using particle swarm theory. In *MHS'95. Proceedings of the sixth international symposium on micro machine and human science* (pp. 39-43). IEEE. doi:10.1109/MHS.1995.494215

Ewees, A. A., Al-qaness, M. A., Abualigah, L., Oliva, D., Algamal, Z. Y., Anter, A. M., Ibrahim, R. A., Ghoniem, R. M., & Abd Elaziz, M. (2021). Boosting arithmetic optimization algorithm with genetic algorithm operators for feature selection: Case study on cox proportional hazards model. *Mathematics*, *9*(18), 2321. doi:10.3390/math9182321

Gong, Y. J., Chen, W. N., Zhan, Z. H., Zhang, J., Li, Y., Zhang, Q., & Li, J. J. (2015). Distributed evolutionary algorithms and their models: A survey of the state-of-the-art. *Applied Soft Computing*, *34*, 286–300. doi:10.1016/j.asoc.2015.04.061

He, Z., Peng, H., Chen, J., Deng, C., & Wu, Z. (2021). A Spark-based differential evolution with grouping topology model for large-scale global optimization. *Cluster Computing*, *24*(1), 515–535. doi:10.1007/s10586-020-03124-z

Heidari, A. A., Mirjalili, S., Faris, H., Aljarah, I., Mafarja, M., & Chen, H. (2019). Harris hawks optimization: Algorithm and applications. *Future Generation Computer Systems*, *97*, 849–872. doi:10.1016/j.future.2019.02.028

Hennessy, J. L., & Patterson, D. A. (2017). *Computer architecture: A quantitative approach*. Elsevier/Morgan Kaufmann.

Holland, J. H. (1992). Genetic algorithms. *Scientific American*, *267*(1), 66–73. doi:10.1038/scientificamerican0792-66

Hussain, K., Mohd Salleh, M. N., Cheng, S., & Shi, Y. (2019). Metaheuristic research: A comprehensive survey. *Artificial Intelligence Review*, *52*(4), 2191–2233. doi:10.1007/s10462-017-9605-z

Ibrahim, R. A., Abualigah, L., Ewees, A. A., Al-Qaness, M. A., Yousri, D., Alshathri, S., & Abd Elaziz, M. (2021). An electric fish-based arithmetic optimization algorithm for feature selection. *Entropy (Basel, Switzerland)*, *23*(9), 1189. doi:10.3390/e23091189 PMID:34573818

Jarray, R., Al-Dhaifallah, M., Rezk, H., & Bouallègue, S. (2022a). Parallel cooperative coevolutionary grey wolf optimizer for path planning problem of unmanned aerial vehicles. *Sensors (Basel)*, *22*(5), 1826. doi:10.3390/s22051826 PMID:35270978

Jarray, R., Yang, B., & Chen, N. (2022b). Arithmetic optimization algorithm based MPPT technique for centralized TEG systems under different temperature gradients. *Energy Reports*, *8*, 2424–2433. doi:10.1016/j.egyr.2022.01.185

Kannan, B. K., & Kramer, S. N. (1994). An augmented Lagrange multiplier based method for mixed integer discrete continuous optimization and its applications to mechanical design. *Transactions of ASME. Journal of Mechanical Design*, *116*(2), 405–411. doi:10.1115/1.2919393

Kaveh, A., & Hamedani, K. B. (2022, January). Improved arithmetic optimization algorithm and its application to discrete structural optimization. *Structures*, *35*, 748–764. doi:10.1016/j.istruc.2021.11.012

Khatir, S., Tiachacht, S., Le Thanh, C., Ghandourah, E., Mirjalili, S., & Wahab, M. A. (2021). An improved Artificial Neural Network using Arithmetic Optimization Algorithm for damage assessment in FGM composite plates. *Composite Structures*, *273*, 114287. doi:10.1016/j.compstruct.2021.114287

Li, L. L., Ren, X. Y., Tseng, M. L., Wu, D. S., & Lim, M. K. (2022). Performance evaluation of solar hybrid combined cooling, heating and power systems: A multi-objective arithmetic optimization algorithm. *Energy Conversion and Management*, *258*, 115541. doi:10.1016/j.enconman.2022.115541

Li, S., Chen, H., Wang, M., Heidari, A. A., & Mirjalili, S. (2020). Slime mould algorithm: A new method for stochastic optimization. *Future Generation Computer Systems*, *111*, 300–323. doi:10.1016/j.future.2020.03.055

Liu, Q., Li, N., Jia, H., Qi, Q., Abualigah, L., & Liu, Y. (2022). A hybrid arithmetic optimization and Golden sine algorithm for solving industrial engineering design problems. *Mathematics*, *10*(9), 1567. doi:10.3390/math10091567

Lu, H. C., Hwang, F. J., & Huang, Y. H. (2020). Parallel and distributed architecture of genetic algorithm on Apache Hadoop and Spark. *Applied Soft Computing*, *95*, 106497. doi:10.1016/j.asoc.2020.106497

Meraihi, Y., Gabis, A. B., Mirjalili, S., & Ramdane-Cherif, A. (2021). Grasshopper optimization algorithm: Theory, variants, and applications. *IEEE Access: Practical Innovations, Open Solutions*, *9*, 50001–50024. doi:10.1109/ACCESS.2021.3067597

Mirjalili, S. (2016). SCA: A sine cosine algorithm for solving optimization problems. *Knowledge-Based Systems*, *96*, 120–133. doi:10.1016/j.knosys.2015.12.022

Mirjalili, S., Mirjalili, S. M., & Lewis, A. (2014). Grey wolf optimizer. *Advances in Engineering Software*, *69*, 46–61. doi:10.1016/j.advengsoft.2013.12.007

Moss, D. R. (2004). *Pressure vessel design manual*. Elsevier.

Premkumar, M., Jangir, P., Kumar, B. S., Sowmya, R., Alhelou, H. H., Abualigah, L., Yildiz, A. R., & Mirjalili, S. (2021). A new arithmetic optimization algorithm for solving real-world multiobjective CEC-2021 constrained optimization problems: Diversity analysis and validations. *IEEE Access: Practical Innovations, Open Solutions*, *9*, 84263–84295. doi:10.1109/ACCESS.2021.3085529

Ragsdell, K. M., & Phillips, D. T. (1976). Optimal design of a class of welded structures using geometric programming. *ASME Journal of Engineering and Industry*, *98*(3), 1021–1025. doi:10.1115/1.3438995

Rahman, M. A., Sokkalingam, R., Othman, M., Biswas, K., Abdullah, L., & Abdul Kadir, E. (2021). Nature-inspired metaheuristic techniques for combinatorial optimization problems: Overview and recent advances. *Mathematics*, *9*(20), 2633. doi:10.3390/math9202633

Storn, R., & Price, K. (1997). Differential evolution–a simple and efficient heuristic for global optimization over continuous spaces. *Journal of Global Optimization*, *11*(4), 341–359. doi:10.1023/A:1008202821328

Wan, L., Gong, K., Zhang, G., Li, C., Wang, Z., & Deng, X. (2021). Ensemble pruning of RF via multi-objective TLBO algorithm and its parallelization on Spark. *IEEE Access: Practical Innovations, Open Solutions*, *9*, 158297–158312. doi:10.1109/ACCESS.2021.3130905

Wang, R. B., Wang, W. F., Xu, L., Pan, J. S., & Chu, S. C. (2021). An adaptive parallel arithmetic optimization algorithm for robot path planning. *Journal of Advanced Transportation*, *2021*, 2021. doi:10.1155/2021/3606895

Wolpert, D. H., & Macready, W. G. (1997). No free lunch theorems for optimization. *IEEE Transactions on Evolutionary Computation*, *1*(1), 67–82. doi:10.1109/4235.585893

Xu, Y. P., Tan, J. W., Zhu, D. J., Ouyang, P., & Taheri, B. (2021). Model identification of the proton exchange membrane fuel cells by extreme learning machine and a developed version of arithmetic optimization algorithm. *Energy Reports*, *7*, 2332–2342. doi:10.1016/j.egyr.2021.04.042

Yang, Y., Qian, C., Li, H., Gao, Y., Wu, J., Liu, C. J., & Zhao, S. (2022). An efficient DBSCAN optimized by arithmetic optimization algorithm with opposition-based learning. *The Journal of Supercomputing*, *78*(18), 1–39. doi:10.1007/s11227-022-04634-w PMID:36247798

Zaharia, M., Xin, R. S., Wendell, P., Das, T., Armbrust, M., Dave, A., Meng, X., Rosen, J., Venkataraman, S., Franklin, M. J., Ghodsi, A., Gonzalez, J., Shenker, S. J., & Stoica, I. (2016). Apache spark: A unified engine for big data processing. *Communications of the ACM*, *59*(11), 56–65. doi:10.1145/2934664

Zhang, Y. J., Yan, Y. X., Zhao, J., & Gao, Z. M. (2022). AOAAO: The hybrid algorithm of arithmetic optimization algorithm with aquila optimizer. *IEEE Access: Practical Innovations, Open Solutions*, *10*, 10907–10933. doi:10.1109/ACCESS.2022.3144431

Zheng, R., Jia, H., Abualigah, L., Liu, Q., & Wang, S. (2021). Deep ensemble of slime mold algorithm and arithmetic optimization algorithm for global optimization. *Processes (Basel, Switzerland)*, *9*(10), 1774. doi:10.3390/pr9101774

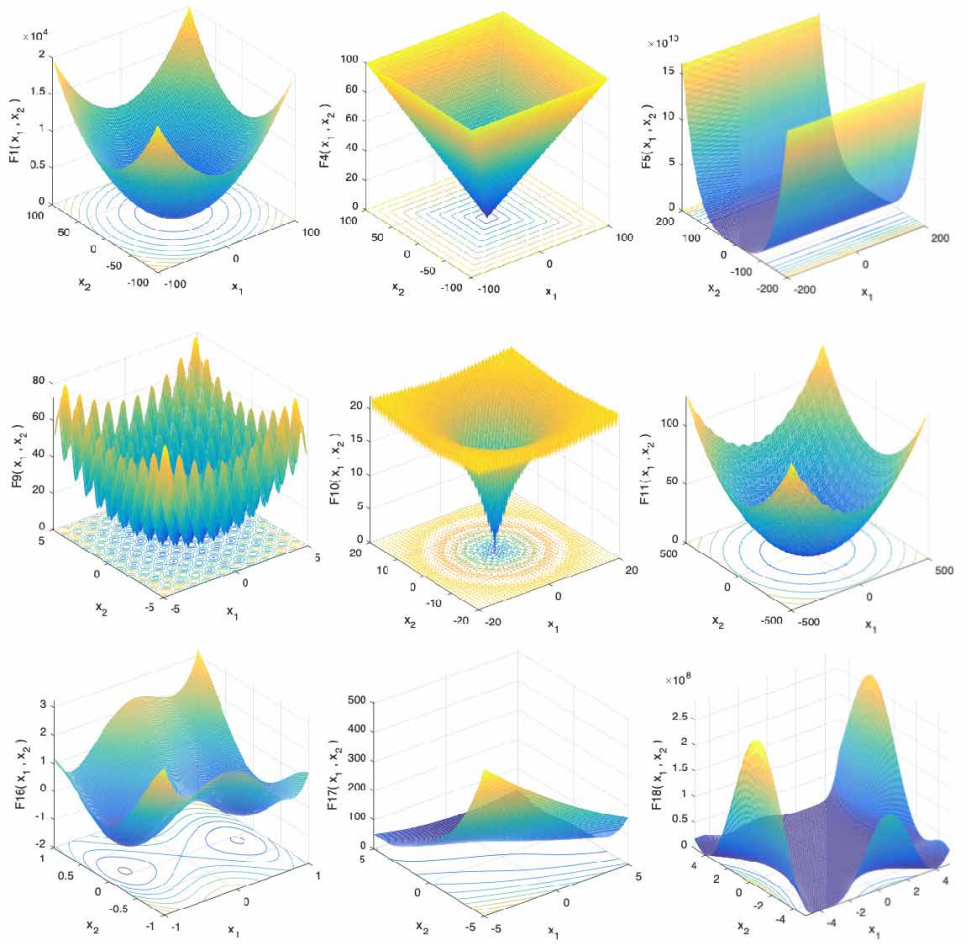
APPENDIX

Table 7 mathematically describes all the benchmark functions used in this paper while Figure 6 illustrates the graphical plot of those functions.

Table 7. Benchmark function details

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	$F_4(x) = max_i \left\{ \lvert x_i \rvert, 1 \le i \le D \right\}$	[-100, 100]	0
Rosenbrock	Unimodal	$F_5(x) = \sum_{i=1}^{D-1} \left[100\left(x_{i+1} - x_i^2\right)^2 + \left(1 - x_i\right)^2 \right]$	[-30, 30]	0
Rastrigin	Multimodal	$F_9(x) = 10D + \sum_{i=1}^{D} \left[x_i^2 - 10\cos\left(2\pi x_i\right) \right]$	[-5.12, 5.12]	0
Ackley	Multimodal	$F_{10}(x) = -20\exp\left(-0.2\sqrt{\frac{1}{D}\sum_{i=1}^{D} x_i^2} \right) - exp(\frac{1}{D}\sum_{i=1}^{D} \cos\left(2\pi x_i\right)) + 20 + e$	[-32, 32]	0
Griewank	Multimodal	$F_{11}(x) = 1 + \frac{1}{4000}\sum_{i=1}^{D} x_i^2 - \prod_{i=1}^{D} \cos\left(\frac{x_i}{\sqrt{i}}\right)$	[-600, 600]	0
CF3	Composite	$F_{16}(CF3)$ $f_1, f_2, f_3, \ldots, f_{10} = Griewank's\ Function$ $[\sigma_1, \sigma_2, \sigma_3, \ldots, \sigma_{10}] = [1,1,1,\ldots,1]$ $[\lambda_1, \lambda_2, \lambda_3, \ldots, \lambda_{10}] = [1,1,1,\ldots,1]$	[-5, 5]	0
CF4	Composite	$F_{17}(CF4)$ $f_1, f_2 = Ackley's\ Function,\ f_3, f_4 = Rastrigin's\ Function,$ $f_5, f_6 = Weierstrass's\ Function,\ f_7, f_8 = Griewank's\ Function,$ $f_9, f_{10} = Sphere's\ Function$ $[\sigma_1, \sigma_2, \sigma_3, \ldots, \sigma_{10}] = [1,1,1,\ldots,1]$ $[\lambda_1, \lambda_2, \lambda_3, \ldots, \lambda_{10}] = \left[\frac{5}{32}, \frac{5}{32}, 1, 1, \frac{5}{0.5}, \frac{5}{0.5}, \frac{5}{100}, \frac{5}{100}, \frac{5}{100}, \frac{5}{100}\right]$	[-5, 5]	0
CF5	Composite	$F_{18}(CF5)$ $f_1, f_2 = Rastrigin's\ Function,\ f_3, f_4 = Weierstrass's\ Function,$ $f_5, f_6 = Griewank's\ Function,\ f_7, f_8 = Ackley's\ Function,$ $f_9, f_{10} = Sphere's\ Function$ $[\sigma_1, \sigma_2, \sigma_3, \ldots, \sigma_{10}] = [1,1,1,\ldots,1]$ $[\lambda_1, \lambda_2, \lambda_3, \ldots, \lambda_{10}] = \left[\frac{1}{5}, \frac{1}{5}, \frac{5}{0.5}, \frac{5}{0.5}, \frac{5}{100}, \frac{5}{100}, \frac{5}{32}, \frac{5}{32}, \frac{5}{100}, \frac{5}{100}\right]$	[-5, 5]	0

Figure 6. Benchmark functions plots



Maryam AlJame earned her B.Sc. and M.Sc. degrees from the Computer Engineering at Kuwait University in 2012 and 2018 respectively. Currently, she is a Teaching Assistant in the Computer Engineering Department at the College of Engineering and Petroleum, Kuwait university. Her research interests include bioinformatics, and parallel and distributed computing.

Aisha Alnoori received her B.Sc. and M.Sc. degrees in Computer Engineering from Kuwait University in 2000 and 2004, respectively. She worked as a computer engineer at State Audit Bureau of Kuwait from 2001-2005. From 2005-2010, she was a teaching assistant at the Information Science Department at Kuwait University. Currently, she is a teaching assistant at the Computer Engineering Department at Kuwait University. Her research interests include embedded systems, artificial intelligence, and network security.

Mohammad G. Alfailakawi obtained his Bachelor of Science (BS) degree in both Electrical Engineering and Computer Engineering from the University of Missouri-Columbia in 1996, and his MS and PhD degrees in Electrical Engineering from the University of Wisconsin-Madison in 1999 and 2002 respectively. He is currently an associate professor at the Computer Engineering Department at Kuwait University where he teaches courses in embedded systems, computer architecture and organization, logic design, Testing, and fault-tolerant computing. Currently, he is the chairman of the computer engineering department at the college of engineering and petroleum, Kuwait University. During 2012 till 2015, he served as the vice dean for academic affairs at the college of Computing Sciences and Engineering. He was also the director of Engineering Training and Alumni center at the College of Engineering and Petroleum at Kuwait University from 2009-2012. His current research interests include design-for-testability, Non-volatile memory test and diagnosis, Defect-based testing, reversible and quantum circuit optimization, metaheuristics, and optimization.

Imtiaz Ahmad received the B.Sc. degree in Electrical Engineering from the University of Engineering and Technology at Lahore, Pakistan, the M.Sc. degree in Electrical Engineering from the King Fahd University of Petroleum and Minerals, Dhahran, Saudi Arabia, and the Ph.D. degree in Computer Engineering from Syracuse University, Syracuse, NY, USA, in 1984, 1988, and 1992, respectively. 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.