Spark Clustering Computing Platform Based Parallel Particle Swarm Optimizers for Computationally Expensive Global Optimization

Qiqi Duan, Lijun Sun, and Yuhui Shi(✉)

Shenzhen Key Laboratory of Computational Intelligence,
Department of Computer Science and Engineering,
Southern University of Science and Technology, Shenzhen 518055, China
shiyh@sustc.edu.cn

Abstract. The increasing demands on processing large-scale data from both industry and academia have boosted the emergence of data-intensive clustering computing platforms. Among them, Hadoop MapReduce has been widely adopted in the evolutionary computation community to implement a variety of parallel evolutionary algorithms, owing to its scalability and fault-tolerance. However, the recently proposed in-memory Spark clustering computing framework is more suitable for iterative computing than disk-based MapReduce and often boosts the speedup by several orders of magnitude. In this paper we will parallelize three mostly cited versions of particle swarm optimizers on Spark, in order to solve computationally expensive problems. First we will utilize the simple but powerful Amdahl's law to analyze the master-slave model, that is, we do quantitative analysis based on Amdahl's law to answer the question on which kinds of optimization problems the master-slave model could work well. Then we will design a publicly available Spark-based software framework which parallelizes three different particle swarm optimizers in a unified way. This new framework can not only simplify the development workflow of Spark-based parallel evolutionary algorithms, but also benefit from both functional programming and object-oriented programming. Numerical experiments on computationally expensive benchmark functions show that a super-linear speedup could be obtained via the master-slave model. All the source code are put at the complementary GitHub project for free access.

Keywords: Parallel particle swarm optimizer · Spark clustering computing Computationally expensive global optimization

1 Introduction

The increasing demands on processing large-scale data from both industry and academia have boosted the emergence of new data-intensive clustering computing platforms. Three of the most successful platforms are the disk-based MapReduce distributed computing paradigm [1–3], the general-purpose GPU heterogeneous computing environment, and the more recently in-memory Spark clustering computing framework [4]. They have been successfully used in a variety of fields (e.g., database [8], machine learning [9], and business intelligence [10]). It is expected by many researchers from the evolutionary computation (EC) community (e.g., [1, 14, 15]) that parallelizing evolutionary algorithms (EAs) on these big data-driven clustering computing platforms could be beneficial to solve computationally expensive problems.

However, designing easy-to-use, scalable, portable, efficient parallel evolutionary algorithms (PEAs) is a non-trivial task. This is mainly due to the fact that we not only need knowledge of hardware architectures and software platforms, but also need to carefully make trade-offs among different performance metrics. For instance, given a fixed number of function evaluations, if a relatively large population size is chosen on each generation for more powerful parallelization, the slower convergence speed may be obtained. On the contrary, if a relatively small population size is chosen for faster convergence, more execution time may be spent. To alleviate such problems, a large number of PEAs based on these big data-driven computing platforms have been proposed (see [14, 15] for comprehensive surveys).

Among them, both Hadoop MapReduce (e.g., [25, 26]) and GPUs (e.g., [6, 7, 14]) have been widely adopted in the EC field to implement a variety of PEAs. To the best of our knowledge, however, there is few work attempting to use Spark, the state-of-the-art in-memory clustering computing platform, to accelerate PEAs. It has been recently found in [4, 9] that Spark is more suitable for iterative computing than MapReduce and often boost the speedup by more than one order of magnitude. Considering significant advantages of Spark over MapReduce, in this paper we will parallelize three highly cited versions of particle swarm optimizer (i.e., PSO [21], CLPSO [11], and ALCPSO [18]) based on Spark, in order to solve computationally expensive problems. More specifically, the main contributions of this paper are two-fold:

1. Inspired by [1, 12], we will utilize the simple but powerful Amdahl's law to theoretically analyze the master-slave model for PEAs, that is, we do quantitative analysis based on the Amdahl's law, in order to answer the question on which kinds of problems the master-slave model could work well (see Sect. 3.2 for more details).
2. We will design a Spark-based software framework parallelizing three highly cited PSOs. This new framework can not only simplify the development workflow of Spark-based PEAs, but also benefit from both functional programming and object-oriented programming (via Scala [22]). The framework is put at the complementary GitHub project[1] for free access. Numerical experiments on computationally expensive test functions show that a *super-linear* speedup could be obtained via the master-slave model.

The rest of the paper is organized as follows. Section 2 gives a brief review of the state-of-the-art works of PEAs. Section 3 describes Spark, the Amdahl's law for the master-salve model, and three Spark-based PSOs. Section 4 conducts numerical experiments. Section 5 gives conclusions and promising research directions.

[1] https://github.com/QiqiDuan257/parallel-pso-spark.

2 Review

In this section, we will review some state-of-the-art works of PEAs, owing to the limit of space. For more comprehensive surveys, please refer to [14, 15, 23].

The most representative work on MapReduce-based PEAs may be the work recently published by Ferrucci et al. [1]. This paper answered a critical research problem regarding when the MapReduce-based PEAs could execute faster than their sequential versions. In [1], a disadvantage of MapReduce-based PEAs (i.e., the overhead caused by communications with the distributed data storage system) was identified.

Wachowiak et al. [6] parallelized a PSO variant in a heterogeneous clustering computing environment, where many-core GPUs are used to run data-parallel operations (e.g., the matrix-matrix multiplication operation) and multi-core CPUs are used to execute other computationally complex tasks (e.g., complicated nested loops). To obtain a higher speedup, they used float-point precision rather than double-point precision in experiments, which may be not suited for real-world applications where high numerical errors are not allowed. Further, the performance of their algorithm depends heavily on the execution profiling to these test functions.

In the cloud computing environment, Zhan et al. [5] proposed a double-layered distributed differential evolution algorithm called Cloudde. The first layer is responsible for operating multiple independent populations with different parameter settings, while the second layer is in charge of computationally intensive function evaluations distributed on multiple virtual machines. The traditional MPI system was applied to realize Cloudde. Although Cloudde showed good performance on some benchmark functions, its scalability and fault-tolerance ability have not yet been tested and thus constitute an open question.

3 Spark-Based Parallel PSOs

This section first compares Spark with other parallel computing technologies. Then Amdahl's law is utilized to quantitatively analyze the master-slave model. Finally, a Spark-based software framework is developed to parallelize three PSOs.

3.1 Comparing Spark with Other Parallel Computing Technologies

Currently, spark is the most active open-source big data project [24]. When compared with MapReduce and MPI, two main advantages of Spark are presented below:

1. It provides a simple yet powerful abstract data structure called resilient distributed dataset (RDD) [20], which can utilize distributed RAM efficiently. Conceptually, RDD can be regarded as an immutable distributed shared memory with implicit data parallelism and fault tolerance. Spark hides the details of hardware architectures and communications among nodes, to some extent. With the help of RDD, developers can focus mainly on the algorithmic logic itself.

2. It supports over 100 high-level operators and the mix of functional programming and object-oriented programming, which simplify the development workflow. For instance, once the function evaluations are finished on different workers, the output can be reduced to the fitness value by invoking the *mapValues* method and then returned to the driver by invoking the *collect* method.

For iterative computation, Spark often reduces the execution time by several orders of magnitude when compared with MapReduce [4, 9].

3.2 Amdahl's Law for the Master-Slave Model

Owing to its simplicity, the master-slave model has been applied to design a variety of PEAs (e.g. Cloudde [5], PEPNet [13]) over two decades. Empirically, it can perform well when the fitness evaluation time dominates the total execution time of the algorithm. However, there is a lack of rigorous quantitative analysis on the theoretical upper bound of the speedup obtained by PEAs based on the master-slave model, except the early work conducted by Dubreuil et al. [12].

In [12], a complicated mathematical model was proposed to analyze the master-slave model, which takes some realistic factors (e.g., communication cost, network latency) into account. However, accurately estimating these parameter values involved is a non-trivial task in practice. Ferrucci et al. [1] hold that the ideal speedup is equal to the cluster size. Strictly speaking, the cluster size is a looser upper bound, when compared with Amdahl's law. Although they mentioned Amdahl's law in their paper, they did not attempt to use it to further analyze PEAs. Their works [1, 12] motivated us to utilize more general Amdahl's law to theoretically and empirically explain ***when and why*** the master-slave model could work well, especially under the Spark clustering computing framework. Inspired by Amdahl's law, we will show in Sect. 4 **when** a ***super-linear*** speedup could be obtained by Spark-based PEAs on computationally intensive continuous benchmark functions.

As stated in Amdahl's law [19, 27], the speedup obtained via parallelization can be calculated according to Eq. 1, as presented below.

$$speedup = \frac{1}{s + \frac{(1-s)}{p}} \tag{1}$$

For Eq. 1, the numerator is the unit time of the sequential program and the denominator is the time spent by the parallel program, where (s) is the serial fraction and (p) is the parallel level. Figure 1 gives a clear description of Amdahl's law, where different serial fractions are considered ranging from 50% to 0.005%.

Under the Spark clustering computing environment, p directly corresponds to the total number of *logical* cores used for function evaluations (rather than the total number of slaves). Therefore, we only need to estimate s for the sequential algorithm, which can be easily done in practice via adding timing. In [27], "*it therefore seems reasonable that there might be a rather even distribution of serial fraction from 0 to 1 over the entire space of computer applications*". We will validate it in the EC field via analyzing several commonly used test functions (see Sect. 4.2 for details).

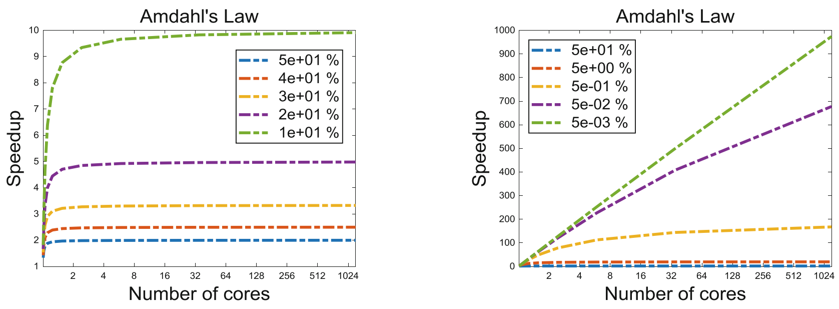

Fig. 1. Amdahl's law [19]. (Different lines correspond to different serial fractions. Note that parallelization may be useful only for highly parallelizable programs [27].)

3.3 Spark-Based PEAs Framework for the Master-Slave Model

In this sub-section, we propose a Spark-based PEAs framework, which can in a unified way parallelize three highly cited PSO versions (i.e., PSO [21], CLPSO [11], and ALCPSO [18]) using the master-slave model. For details of these three PSOs, please refer to their corresponding original papers. For their concrete implementation details, please refer to the Scala source code on the complementary GitHub project.

This Spark-based PEAs framework is built on a unified interface with three basic configuration classes and an algorithm base class as parameters. Although sequential algorithms are also supported in this framework, we focus mainly on the parallelization of population-based evolutionary algorithms. Three configuration classes are *ConFuncParams*, *TestParams*, and *AlgoParams*, respectively. The *ConFuncParams* class includes the function name, function dimension, upper and lower search bounds during optimization, and initial upper and lower search bounds at the beginning stage of the search. The *TestParams* class includes the total number of independent tests, and random seeds to initialize the population. The *AlgoParams* class includes the population size, and the maximum of function evaluations, which can be inherited to customize the parameter settings. All algorithm sub-classes inherited from the algorithm base class have the same method called *optimize*, which takes as input the function, and as output the final optimization results. Taking as inputs the function rather than reference or value is one of very useful and flexible characteristics for functional programming. The unified interface takes as input two functions, one of which is the method of the optimization algorithm (i.e., *optimize*) and another of which is the function optimized at hand. Such functional programing-based design increases the scalability and flexibility of the proposed PEAs framework.

To parallelize function evaluations, a simple but resilient data structure built in Spark (i.e., *RDD*) is used. First we use the parallelize method of the built-in *SparkContext* object to transfer all individuals from the master to slave nodes. For simplicity, the parallel level is equal to the population size. Then function evaluations tasks can be started by invoking the built-in *mapValues* method. Finally, all the fitness values are returned from different slave nodes to the master by invoking the built-in *collect* method. For more details, please refer to the public Scala source code. Overall, fulfilling the master-slave model for PEAs is simple and straight under the Spark clustering computing framework.

4 Numerical Experiments

In this section, we first describe a private Spark clustering computing platform used here. Then five of most commonly used continuous benchmark functions are empirically analyzed according the Amdahl's law. Finally, comparisons between sequential and parallel PSOs are conducted.

4.1 The Spark Clustering Computing Platform

All numerical experiments were conducted on a private Spark clustering computing platform with a total of 160 CPU cores, which consists of a master node (i.e., the driver) and three slave nodes (i.e., the workers). Except that the master node has four 480-GB SSD hard disks working in RAID 1+0 for high-availability, all the nodes have the same hardware and software configurations, as presented in Table 1. The recommendations from the Spark official website [16] are followed to configure the hardware. We also give a practical guidance on the online appendix[2] to illustrate how to rapidly and efficiently deploy a private Spark clustering computing platform. Both Matlab and Scala are also installed on these machines to run sequential algorithms. For Scala, the third-party numerical processing library (i.e., *breeze* [17]) is used.

Table 1. Hardware and software configurations for each node.

Hardware	Setting	Software	Version
Machine	Dell® PowerEdge R730 Server	OS	CentOS 7.3.1611
Architecture	64-bit	Spark	2.2.0
CPU	40 Intel® Xeon E5-2640 v4 @ 2.40 GHz	Scala	2.11.11
RAM	64 GB	Sbt	1.0.1
Hard disk	A 960 GB SSD hard disk *without* RAID	Matlab	R2016b (glnxa64)
Network	1Gbps LAN	Java	1.8.0_131

4.2 Analyses of Continuous Benchmark Functions

To compare the performance of different algorithms, five well-known continuous benchmark functions (i.e., Sphere, Rosenbrock, Rastrigin, Griewank, and Schwefel12) [18] are used. Because they have different landscape characteristics (e.g., unimodal vs. multimodal, and no-separable vs. separable) and different time complexities (e.g., linear vs. quadratic), we can compare their run time on different scenarios.

[2] https://github.com/QiqiDuan257/parallel-pso-spark.

To test the performance of PEAs on computationally expensive problems, a common practice is to use high-dimensional benchmark functions. However, we found that some high-dimensional benchmark functions may be not computationally expensive, assuming that for computationally expensive functions the function evaluations time should dominate the total execution time. According to the proportion of the function evaluations (i.e., FEs) time, these five high-dimensional benchmark functions can be classified empirically into two categories, as presented below:

1. Benchmark functions with a low proportion of the FE time include Sphere, Rosenbrock, Rastrigin, and Griewank. All of them have a *linear* time complexity with the dimension. As we can see from Fig. 2, for PSO, CLPSO, and ALCPSO, almost all of the proportions of FE are less than 50% even when the dimension reaches 1e7. According to Amdahl's law, we can predict that the master-slave model could only obtain a limited speedup on these functions, which is less than 2 even in the ideal case. In the following parts, we will further validate our aforementioned prediction in Spark.
2. Benchmark functions with a high proportion of the FE time on high dimension include Schwefel12 with a quadratic time complexity. As shown in Fig. 3, when the dimension exceeds 1e3, the proportion of the FE time reaches more than 95%. Based on Amdahl's law, it can be theoretically estimated that the master-slave model could show a significant speedup on this function. In the following parts, we will further prove that even a *super-linear* speedup can be achieved on this function in Spark.

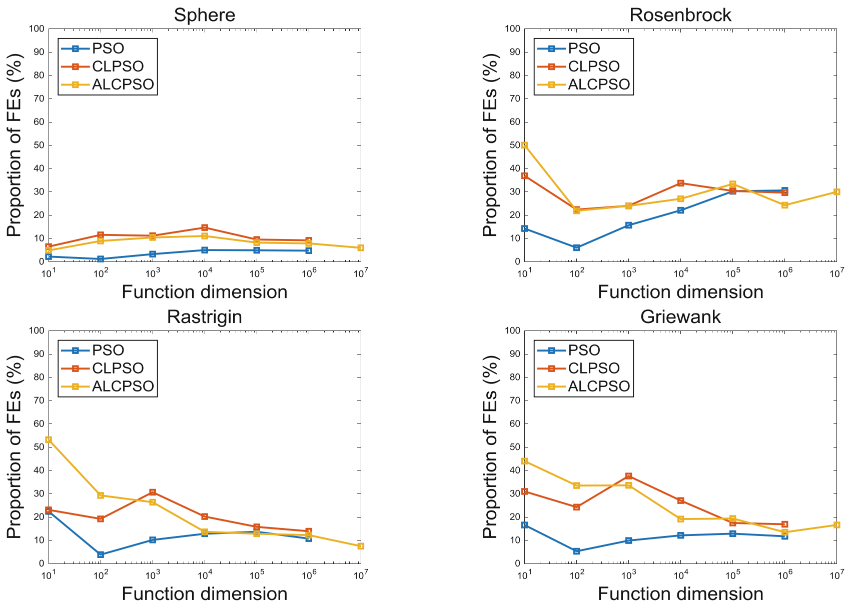

Fig. 2. Four benchmark functions with a low proportion of the function evaluations time varying with function dimensions for PSO, CLPSO, and ALCPSO.

When using a PEA based on the master-slave model, we may first calculate the proportion of the FE time on its sequential version, and then estimate the theoretical speedup through Amdahl's law. In most cases this speedup may be ***over-estimated*** owing to a variety of overheads in practice (e.g., communication cost, synchronization barriers, and network latency). However, it is worth noting that we still achieve a *super-linear* speedup in some cases, often caused by *strong scaling* [27].

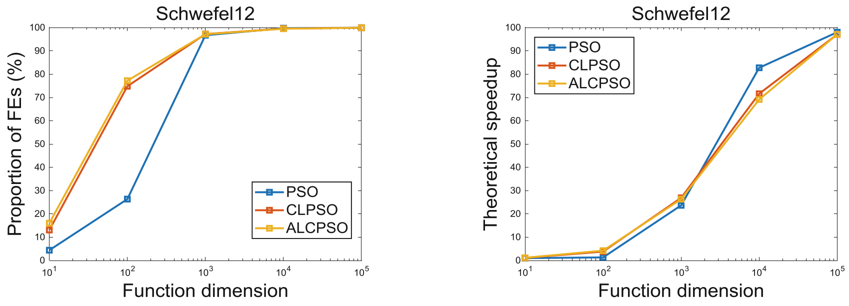

Fig. 3. A benchmark function with a high proportion of the function evaluations time.

4.3 Comparisons on Computationally Expensive Functions

We first compare three Spark-based PSOs with their corresponding sequential versions on the computationally intensive Schwefel12 benchmark function varying function dimensions from 1e1 to 1e5. To reduce statistical errors, all numerical experiments were run independently 30 times (except for inefficient sequential versions), and the average run time was recorded, as shown in Fig. 4. To make fair comparisons, for all algorithms, the population size and the maximum of function evaluations are set to 100 and 500, respectively. For high-dimensional problems, a relatively large population size (e.g., 100) is preferred to enhance exploration. Because the total run time of all the sequential algorithms on high dimensions is unaccepted for the large number of FE, a relatively small number of FE (i.e., 500) is used here. Other parameter settings of all algorithms follow the suggestions given in their corresponding original papers. Considering the repeatability of the experiment, all data and source code are freely available on the complementary GitHub project.

As we can see from Fig. 4, all three Spark-based PSOs can obtain the significant speedup on high dimensions, when compared with their corresponding Matlab-based sequential versions. More specifically, on 1e3, 1e4, and 1e5 dimensions, Spark-based PSO, CLPSO and ALCPSO achieve the (3x, 41x, 224x), (6x, 50x, 194x), and (5x, 44x, 184x) speedup, respectively. However, on 10 and 100 dimensions, since the communications overheads between the master and all slaves cancel out the speedup obtained via parallelization, even worse results are obtained.

To test the scalability of the proposed algorithms on the function with the 1e5 dimension, we linearly increased the maximum of FE from 1000 to 5000 with step 1000. To reduce statistical errors, all numerical experiments were run independently 30 times for all three Spark-based parallel PSOs (except for inefficient sequential contenders), and the average run time was summarized, as presented in Fig. 5. It can be observed from Fig. 5 that all three parallel PSOs can obtain the *super-linear* speedup on this high-dimensional, computationally expensive function. On the contrary, the time complexities of all three Matlab-based sequential versions linearly rise with the number of FE. For parallel PSOs, some stability issues raise with the increasing number of FE, which will be analyzed in Fig. 6.

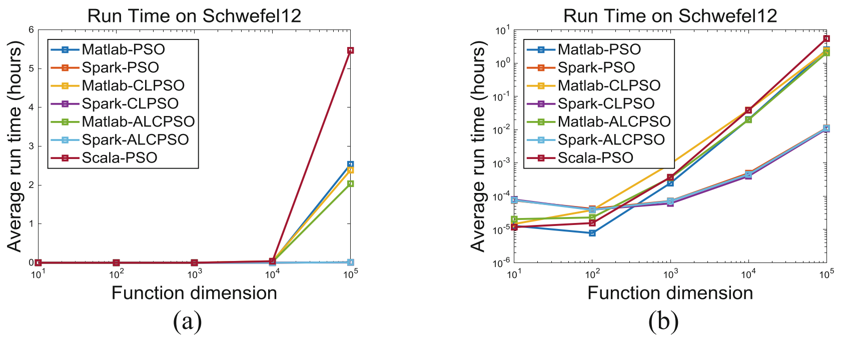

Fig. 4. Comparisons of run time for Three Spark-based parallel PSOs *versus* sequential counterparts on Schwefel12 varying with function dimensions. (Since some lines are condensed into one single line in the left figure (a) owing to the large magnitude of the y-axis, we enlarge them in the right figure (b) via logarithmizing the *y*-axis.)

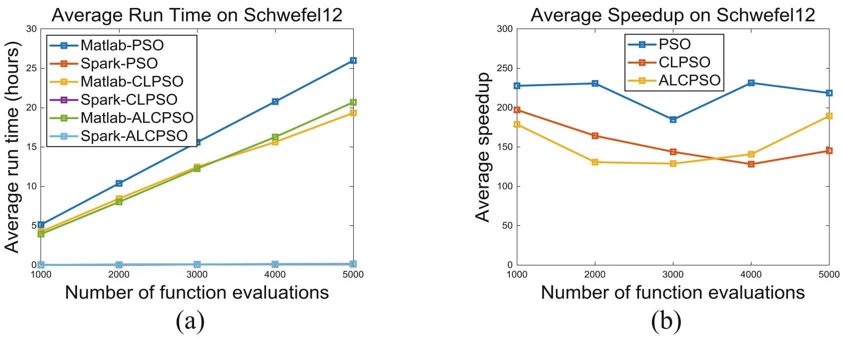

Fig. 5. Comparisons of speedup for Spark-based parallel PSOs *versus* Matlab-based sequential counterparts on 100000-dimensional Schwefel12 varying with number of function evaluations. (Note that some lines are condensed into one single line in the left figure (a) owing to the large magnitude of the *y*-axis.)

To further analyze the stability (i.e., fault-tolerance ability) of the proposed parallel algorithms, we plotted the boxplots of the execution time for all three Spark-based PSOs in Fig. 6. We can see that there are some outliers, which take approximately up to 3x times than typical runs. Although more time is spent, the program could automatically be recovered from the struggling state which may be caused by the underlying network instability. In fact, the good fault-tolerance ability of Spark has been empirically proven in industry [4], which is one advantage over MPI in practice.

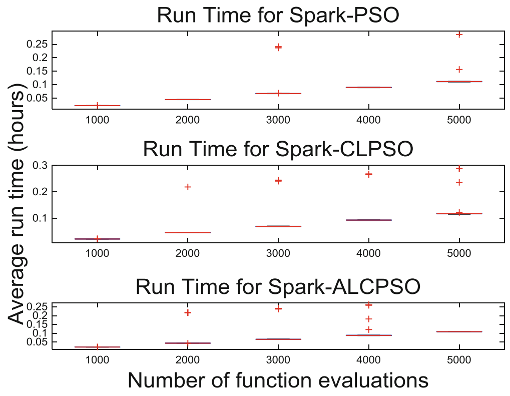

Fig. 6. Boxplot of the execution time obtained on 30 independently runs.

4.4 Comparisons on Functions with Linear Time Complexity

We conducted experiments on four high-dimensional yet computationally-cheap benchmark functions. All experiments were run independently 30 times. For all four functions, the dimension and maximum of FE are set to 1e5 and 500, respectively. For all algorithms used here, the population size is set to 100.

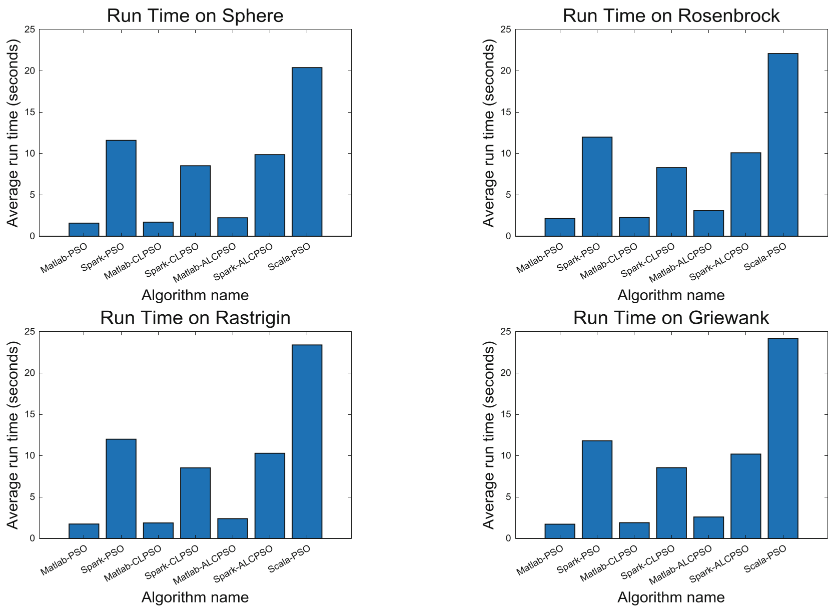

Fig. 7. Comparisons of run time on four computationally-cheap benchmark functions.

As we can see from Fig. 7, three Spark-based parallel PSOs do not obtain any speedup on computationally-cheap benchmark functions, when compared with their corresponding sequential counterparts. This is due to the fact that the communication and synchronization costs among the master and all slaves heavily exceed the parallelization benefit. The "*one-size-fits-all*" parallelization strategy may not exist.

5 Conclusions and Future Research Directions

In this paper we first analyzed the speedup of PEAs using the master-slave model. According to Amdahl's law, we pointed out *when* the master-slave model could work well. Then we provided a Spark-based PEAs framework based on which three most cited PSOs have been parallelized using the master-slave model. The experimental results showed that a super-linear speedup could be obtained by the proposed parallel PSOs at least on computationally expensive test functions. However, there are some open questions which are our future research directions and are presented below:

1. The effectiveness and efficiency of the proposed PEAs need to be further tested on more realistic optimization problems (e.g. geostatic correction [6]).
2. For data-intensive function evaluations tasks, how do Spark-based PEAs read data from the distributed file storage system efficiently?

Acknowledgements. This work is partially supported by the Ministry of Science and Technology (MOST) of China under the Grant No. 2017YFC0804002, National Science Foundation of China under the Grant No. 61761136008, and Science and Technology Innovation Committee Foundation of Shenzhen under the Grant No. ZDSYS201703031748284. We acknowledge three anonymous reviewers for their valuable comments and Dr. Jun Huang, Hao Tong, Chang Shao, Liang Qu, and Jing Liu for their help.

References

1. Ferrucci, F., Salza, P., Sarro, F.: Using Hadoop MapReduce for parallel genetic algorithms: a comparison of the global, grid and island models. Evol. Comput. **29**, 1–33 (2018). Early Access
2. Dean, J., Ghemawat, S.: MapReduce: simplified data processing on large clusters. Commun. ACM **51**(1), 107–113 (2008)
3. Dean, J., Ghemawat, S.: MapReduce: a flexible data processing tool. Commun. ACM **53**(1), 72–77 (2010)
4. Zaharia, M., Xin, R.S., Wendell, P., et al.: Apache Spark: a unified engine for big data processing. Commun. ACM **59**(11), 56–65 (2016)
5. Zhan, Z.H., Liu, X.F., Zhang, H., et al.: Cloudde: a heterogeneous differential evolution algorithm and its distributed cloud version. IEEE Trans. Parallel Distrib. Syst. **28**(3), 704–716 (2017)
6. Wachowiak, M.P., Timson, M.C., DuVal, D.J.: Adaptive particle swarm optimization with heterogeneous multicore parallelism and GPU acceleration. IEEE Trans. Parallel Distrib. Syst. **28**(10), 2784–2793 (2017)

7. Kan, G., Lei, T., Liang, K., et al.: A multi-core CPU and many-core GPU based fast parallel shuffled complex evolution global optimization approach. IEEE Trans. Parallel Distrib. Syst. **28**(2), 332–344 (2017)
8. Thusoo, A., Sarma, J.S., Jain, N., et al.: Hive: a warehousing solution over a map-reduce framework. Proc. VLDB Endow. **2**(2), 1626–1629 (2009)
9. Meng, X., Bradley, J., Yavuz, B., et al.: MLlib: machine learning in Apache Spark. J. Mach. Learn. Res. **17**(1), 1235–1241 (2016)
10. Armbrust, M., Xin, R.S., Lian, C., et al.: Spark SQL: relational data processing in Spark. In: Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, pp. 1383–1394. ACM (2015)
11. Liang, J.J., Qin, A.K., Suganthan, P.N., et al.: Comprehensive learning particle swarm optimizer for global optimization of multimodal functions. IEEE Trans. Evol. Comput. **10** (3), 281–295 (2006)
12. Dubreuil, M., Gagné, C., Parizeau, M.: Analysis of a master-slave architecture for distributed evolutionary computations. IEEE Trans. Syst. Man Cybern. Part B (Cybern.) **36**(1), 229–235 (2006)
13. Riessen, G.A., Williams, G.J., Yao, X.: PEPNet: parallel evolutionary programming for constructing artificial neural networks. In: Angeline, P.J., Reynolds, R.G., McDonnell, J.R., Eberhart, R. (eds.) EP 1997. LNCS, vol. 1213, pp. 35–45. Springer, Heidelberg (1997). https://doi.org/10.1007/BFb0014799
14. Tan, Y., Ding, K.: A survey on GPU-based implementation of swarm intelligence algorithms. IEEE Trans. Cybern. **46**(9), 2028–2041 (2016)
15. Gong, Y.J., Chen, W.N., Zhan, Z.H., et al.: Distributed evolutionary algorithms and their models: a survey of the state-of-the-art. Appl. Soft Comput. **34**, 286–300 (2015)
16. Spark Hardware Provisioning Homepage. http://spark.apache.org/docs/latest/hardware-provisioning.html. Accessed 02 Apr 2018
17. Scala Breeze Homepage. https://github.com/scalanlp/breeze. Accessed 02 Apr 2018
18. Chen, W.N., Zhang, J., Lin, Y., et al.: Particle swarm optimization with an aging leader and challengers. IEEE Trans. Evol. Comput. **17**(2), 241–258 (2013)
19. Kirkpatrick, K.: Parallel computational thinking. Commun. ACM **60**(12), 17–19 (2017)
20. Zaharia, M., Chowdhury, M., Das, T., et al.: Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing. In: Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation, p. 2. USENIX Association (2012)
21. Shi, Y., Eberhart, R.: A modified particle swarm optimizer. In: IEEE World Congress on Computational Intelligence, pp. 69–73 (1998)
22. Odersky, M., Spoon, L., Venners, B.: Programming in Scala. Artima Inc., Mountain View (2016)
23. Alba, E., Tomassini, M.: Parallelism and evolutionary algorithms. IEEE Trans. Evol. Comput. **6**(5), 443–462 (2002)
24. Spark GitHub Homepage. https://github.com/apache/spark. Accessed 02 Apr 2018
25. Verma, A., Llorà, X., Goldberg, D.E., et al.: Scaling genetic algorithms using MapReduce. In: Ninth International Conference on Intelligent Systems Design and Applications, pp. 13–18. IEEE (2009)
26. Hajeer, M.H., Dasgupta, D.: Handling big data using a data-aware HDFS and evolutionary clustering technique. IEEE Trans. Big Data (2017). Early Access
27. Gustafson, J.L.: Amdahl's law. In: Padua, D. (ed.) Encyclopedia of Parallel Computing, pp. 53–60. Springer, Boston (2011). https://doi.org/10.1007/978-0-387-09766-4