

A Parallel Version of Differential Evolution Based on Resilient Distributed Datasets Model

Changshou Deng^(✉), Xujie Tan, Xiaogang Dong, and Yucheng Tan

School of Information Science and Technology, JiuJiang University,
Jiujiang 332005, Jiangxi, China
{csdeng,txj2010,xx.dongxiaogang,yctan}@jju.edu.cn

Abstract. MapReduce is a popular cloud computing platform which has been widely applied in large-scale data-intensive fields. However, when dealing with computation extensive tasks, particularly, iterative computation, frequent loading Map and Reduce processes will lead to overhead. Resilient distributed datasets model which has been implemented in Spark, is an in-memory clustering computing which can overcome this shortcoming efficiently. In this paper, we attempt to use resilient distributed datasets model to parallelize Differential Evolution algorithm. A wide range of benchmark problems have been adopted to conduct numerical experiment, and the speedup of PDE due to use of resilient distributed datasets model is demonstrated. The results show us that resilient distributed datasets model is a potential way to parallelize evolutionary algorithm.

Keywords: Parallel differential evolution · Spark · Resilient distributed datasets · Transformation operation · Action operation

1 Introduction

Evolutionary algorithms (EAs) have been successfully applied in solving numerous optimization problems in diverse fields. Among them, Differential evolution (DE) algorithm is a simple powerful population-based stochastic search technique, which is an efficient and effective global optimizer in the continuous field [1]. Comparing with classical EAs such as Genetic Algorithm (GA), Evolutionary Strategy (ES), and the Swarm Intelligence Optimization algorithm i.e. particle swarm optimization (PSO), it has been claimed that DE exhibited an overall excellent performance for a wide range of benchmark problems [2]. Since its inception, DE has been applied to many real-world problems successfully [3, 4].

Inspired by the great success of the classic DE, numerous variants of DE have been developed for solving different types of optimization problems such as noisy, constrained, and dynamic optimization problems. Recently, several enhanced DE has been proposed to improve the performance of DE [5–7]. However, in many engineering applications, each evaluation of the quality of solution is very time consuming. The use of parallel computing is a remedy in reducing the computing

time required for complex problems. Due to DE maintaining a lot of individuals in the population, DE has an implicit parallel and distributed nature. Therefore, several parallelization techniques of EAs have been reported [8]. Actually, a parallel implementation of multi-population DE has been proposed with parallel virtual machine [9]. Recently, Graphics Processing Unit (GPU) was used to implement parallel DE [10, 11]. MapReduce is a programming model which was originally designed to simplify the development of distributed application for large scale data processing by Google [12]. There have been several attempts at using MapReduce model to parallelize EAs [13–16]. However, EAs are iterative algorithms working in loops, with output of each iteration being input for next iteration. By contrary, MapReduce is designed to run only once and produce final outputs immediately. Thus, parallelizing EAs with MapReduce leads to restart a MapReduce process during each generation of EAs. Frequent calling MapReduce process will increase much overhead. Previous works have proved that overhead decreased performance gaining from adding new nodes [16].

In this paper, a parallel implementation of DE based on resilient distributed datasets (RDD) [17] model is proposed. RDD is a distributed memory abstraction that allows programmers to perform in-memory computations on large clusters while retaining the fault tolerance of data flow models as MapReduce [19]. RDD supports iterative operations and interactive data mining. To overcome the shortcoming of parallelized DE with MapReduce, we parallelize DE using RDD.

The remainder of the paper is organized as follows. Section 2 describes the conventional DE. Resilient distributed datasets (RDD) model is presented in Sect. 3. With RDD model, the parallel implementation of DE (PDE) is proposed in Sect. 4. Comparing with DE, the performance of PDE is evaluated through numerical experiment in Sect. 5. Finally, Sect. 6 concludes this paper.

2 Differential Evolution

DE is a heuristic approach for minimizing continuous optimization problem which is possibly nonlinear and non-differentiable. DE maintains a population of D -dimensional vectors and requires few control variables. It is robust, easy to use and lends itself very well to parallel computation. The four operations, namely initialization, mutation, crossover and selection, in classical DE [1], are given as follows. Initialization in DE is according to Eq. (1).

$$x_{ij}(G) = x_j^l + rand_j(0, 1)(x_j^u - x_j^l). \quad (1)$$

Where $G = 0, i = 1, 2, \dots, NP, j = 1, 2, \dots, D, x_j^u$ denotes the upper constraints, and x_j^l denotes the lower constraints.

After being initialized, for each target vector $X_{i,G}, i = 1, 2, \dots, NP$, a mutant vector is produced according to Eq. (2)

$$v_{i,G+1} = x_{r1,G} + F(x_{r2,G} - x_{r3,G}). \quad (2)$$

where $i, r1, r2, r3 \in \{1, 2, \dots, NP\}$ are randomly chosen and have to be mutually exclusive. And F is the scaling factor for the difference between the individual x_{r2} and x_{r3} .

In order to increase the diversity of population, DE introduces the crossover operation to generate a trial vector which is the mixture of the target vector and the mutation vector. In traditional DE, the uniform crossover is defined as follows:

$$u_{i,G+1} = \begin{cases} v_{i,G+1} & \text{if } rand \leq CR \text{ or } j = rand(i) \\ x_{i,G} & \text{otherwise} \end{cases} \quad (3)$$

where $i = 1, 2, \dots, NP, j = 1, 2, \dots, D, CR \in [0, 1]$ is the crossover probability and $rand(i) \in (0, 1, 2, \dots, D)$ is the randomly selected number which ensures that the trial vector $(u_{i,G+1})$ gets at least one element from the mutation vector $(v_{i,G})$.

To decide which one will survive in the next generation, the target vector $(x_{i,G})$ is compared with the trial vector $(u_{i,G+1})$ in terms of objective value according to

$$x_{i,G+1} = \begin{cases} u_{i,G+1}, & \text{if } f(u_{i,G+1}) < f(x_{i,G}) \\ x_{i,G} & \text{otherwise.} \end{cases} \quad (4)$$

3 Resilient Distributed Datasets (RDD)

Cloud computing represents a pool of virtual resources for information processing. High level cloud computing models like MapReduce [12] and Dryad [18] have been widely used to process the growing big data. These computing cluster systems are based on an acyclic data flow model which does not support for working set. Thus the applications based on an acyclic data flow model have to write data to disk and reload it on each iteration operation with current systems, leading to significant overhead. RDD allow programmers to explicitly cache working sets in memory across iteration operation, leading to substantial speedups on future use.

3.1 RDD Abstraction

RDD provides an abstraction that supports applications with working set. RDD not only supports data flow models, but also be capable of efficiently expressing computations with working sets. During operation on a working set, RDD only supports coarse-grained transformations, where a single operation can be applied to many records. Formally, an RDD is a collection of elements partitioned across the nodes of the cluster that can be operated on in parallel. RDDs can be created only through two ways: (1) either by starting with an existing file in stable storage, or (2) by an existing Scala collection in the driver program and transforming it.

3.2 Programming Model in Spark

Spark is the first system allowing an efficient, general purpose programming language to be used interactively to analyze datasets on clusters [17]. In Spark, RDDs are represented by objects, and transformations are invoked using methods on these objects. After defining one or more RDDs, transformation operations are used to transform these RDDs. Then action operations are used to return value to driver program or export data to disk storage. The programming model is presented in Fig. 1.

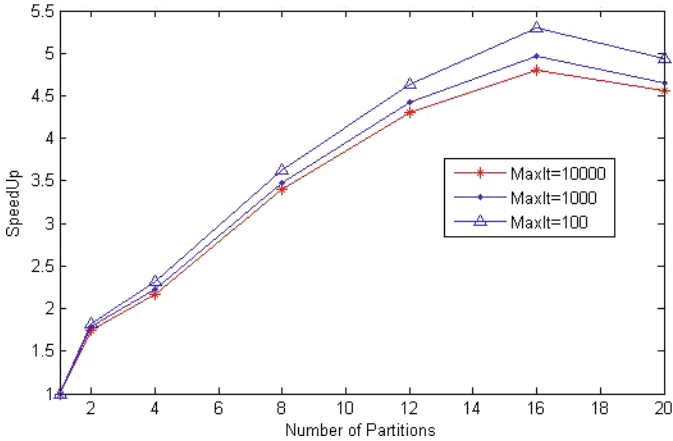


Fig. 1. RDD model in spark

3.3 RDD Operations in Spark

RDD includes mainly three types of operations: transformation operations, control operations and action operations. Transformations create a new dataset from an existing one, and control operations can persist an RDD in memory with cache method, in such case Spark will keep the elements around on the cluster for much faster access the next time you query it. Action operations return a value to the driver program after running a computation on the dataset.

4 Parallel DE

4.1 Procedure of PDE

RDD is a fault-tolerant abstraction for in-memory cluster computing. With the transformations available in Spark, a parallel version of DE is proposed. For many optimization problems, the evaluation of their objective function is costly.

Thus, in our proposed PDE, we only use cluster computing to compute the fitness values of the individuals. The Steps of the PDE is depicted as follows.

Algorithm. PDE

```

Input: NP; F; CR; MaxIt(Maximum number of iterations); 1: Randomly
initialize population P with NP individuals 2: Creation of RDD 3:
Evaluate the objective function value f for each individuals
  in parallel way
4: generation=0
5: while generation < MaxIt do
6: for i=1:NP
7: Mutation operation according to Eq.(2)
8: Crossover operation according to Eq.(3)
9: Creation of RDD for mutation operation U
10: Evaluate the objective function value f for each
  individuals (U) in parallel way
11: Selection operation according to Eq.(4)
12:end for
13:generation=generation+1
14: end while
Output: the best individual with smallest fitness

```

The core of Spark is implemented in Scala language. Thus we follow Spark to use Scala language to implement PDE. Our example codes for evaluate the objective function in parallel way are as follows:

```

val point = sc.parallelize(pop,numSlices).cache()
var popf = points.map(
  p =>(p.x,getFitness(p.y,ifun+1),p.y)).collect()

```

In the codes, the first line is the process of creation of RDD from population individuals denoted by *pop*, and the second line is the demonstration of map transformation and collect action to conduct the evaluation of objective function in parallel way.

4.2 Inspection of PDE

First of all, EAs spend the majority of the computational time for evaluating the objective function values when applied to real-world applications. With RDD, the proposed PDE distributes the objective function evaluation to predefined partitions. Then, all individuals in PDE can be evaluated in parallel way. Consequently, the proposed PDE can be regarded as an efficient program. Comparing DE with PDE, there is no significant difference in the procedure. Therefore, PDE can be regarded as an efficient algorithm. Furthermore, the steps of PDE can be implemented by Scala, Java, or Python programming language. Therefore, the proposed PDE is portable.

5 Numerical Experiment

5.1 Benchmark Problems

In order to evaluate the performance of PDE, the benchmark problems used in this paper are listed in Table 1. Functions f_1 and f_2 are unimodal, while functions f_3, f_4, f_5 and f_6 are multimodal. In our experiment, all the benchmark problems have $D = 30$ dimensional real-parameters.

Table 1. Benchmark problems

Name	Expression	Value Range
Sphere function	$f_1 = \sum_{i=1}^D x_i^2$	[-100,100]
Schwefel's Ridge function	$f_2 = \sum_{i=1}^D (\sum_{k=1}^j x_k)^2$	[-100,100]
Rosenbrock function	$f_3 = \sum_{j=1}^{D-1} (100(x_{j+1} - x_j^2)^2 + (x_j - 1)^2)$	[-30,30]
Rastrigin function	$f_4 = \sum_{j=1}^D (x_j^2 - 10\cos(2\pi x_j) + 10)$	[-5.12,5.12]
Ackley function	$f_5 = -20 \exp(-0.2\sqrt{\frac{1}{D} \sum_{j=0}^D x_j^2} - \exp(\frac{1}{D} \sum_{j=1}^D \cos(2\pi x_j)))$	[-32,32]
Griewank function	$f_6 = \frac{1}{4000} \sum_{j=1}^D x_j^2 - \prod_{j=1}^D \cos(\frac{x_j}{\sqrt{j}}) + 1$	[-600,600]

5.2 Experimental Results

PDE and DE are applied to the six benchmark problems. The setting of parameters used in PDE and DE are $NP = 10 * D, F = 0.5, CR = 0.9$ and $MaxIT = 10000$. Twenty independent runs are carried out for the two algorithms in each function. In our experiment, Dell computers with 3.4 Ghz Intel Core i7-3770 CPU and 8G of RAM are used to construct the computing cluster. Spark1.2.0 is adopted as experimental platform. In PDE, we choose four different numbers of partition of RDD, namely, 2, 4, 8, and 15. Table 2 shows the objective function values of the best solutions obtained by PDE with different partition number, and DE.

In order to evaluate the speedup of the proposed PDE effectively, we add some delay in each objective function. The speedup metric mentioned in [19] was used in this paper. The speedup of PDE is defined as follows:

$$S_m(N_p) = \frac{T_m(1)}{T_m(N_p)} \quad (5)$$

In Eq. (5) $T_m(1)$ denotes the execution time of DE averaged over m times with one partition, while $T_m(N_p)$ denotes the averaged execution time of the proposed PDE achieved with N_p partitions in RDD. In this paper, $m = 5$.

The speedup curves achieved by the proposed PDE for the six benchmark problems are plotted in from Figs. 2, 3, 4, 5, 6 and 7 respectively.

Table 2. Objective function value

Function	PDE				DE
	(2)	(4)	(8)	(15)	
f_1	$7.80E - 073$	$4.30E - 073$	$1.55E - 072$	$3.98E - 073$	$1.80E - 071$
f_2	$6.48E - 014$	$5.92E - 014$	$5.36E - 014$	$6.61E - 014$	$8.43E - 012$
f_3	$0.00E + 000$	$0.00E + 000$	$0.00E + 000$	$0.00E + 000$	$0.00E + 000$
f_4	$9.74E + 001$	$1.16E + 002$	$1.26E + 002$	$1.08E + 002$	$1.43E + 002$
f_5	$4.00E - 015$	$4.00E - 015$	$4.00E - 015$	$4.00E - 015$	$4.44E - 015$
f_6	$0.00E + 000$	$0.00E + 000$	$0.00E + 000$	$0.00E + 000$	$0.00E + 000$

5.3 Discussion of Experimental Results

From Table 2, there is not a significant difference between PDE, and DE in the quality of solutions for functions f_3 and f_6 . For the remainder two functions f_1 , f_2 , f_4 and f_5 , the results of PDE are slightly better than those of DE.

From the speedup curves shown in Figs. 2, 3, 4, 5, 6 and 7, we can confirm that the speedup is larger than one in every instance. Therefore, the proposed PDE reduces the computational time with different numbers of partitions. The speedup achieved by PDE increases as the number of partitions increases steady

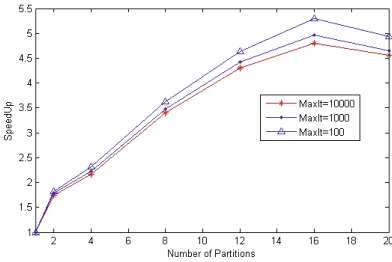


Fig. 2. Speedup by PDE on function: f_1

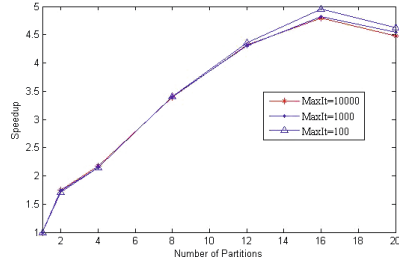


Fig. 3. Speedup by PDE on function: f_2

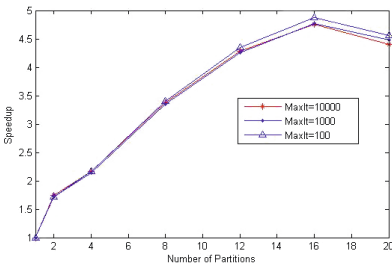


Fig. 4. Speedup by PDE on function: f_3

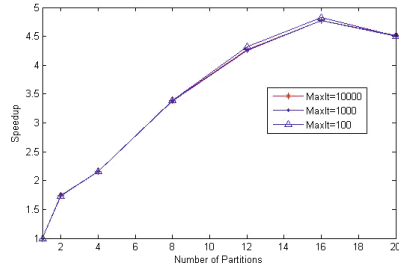


Fig. 5. Speedup by PDE on function: f_4

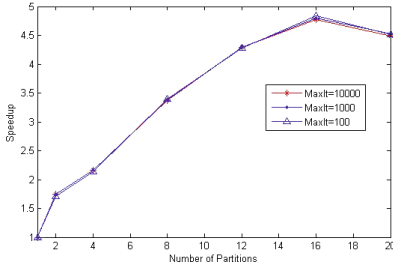


Fig. 6. Speedup by PDE on function: f_5

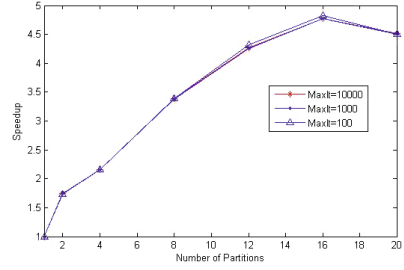


Fig. 7. Speedup by PDE on function: f_6

until $N_p = 16$ in all benchmark problems. The speedup will decrease when the number of partitions is larger than 16. Spark runs one task for each partition of the cluster. When more partitions are involved, the communication cost between nodes will decrease the speedup. There is no significant difference in speedup between the three maximum iterations(MaxIt) for each instance. The speedup of PDE actually depends on the cost of evaluate the objective function. We can expect that the proposed PDE is useful specially for solving the real-world applications that spend the majority of the computational time for evaluating their objective function values.

6 Conclusion

In order to utilize the cloud computing platform to parallelize DE, Spark, an open source cloud computing platform which supports iterative computation, was adopted. The proposed PDE was based on resilient distributed datasets model. In our PDE, the computation of objective function was parallelized. Therefore, we could expect the computational time was reduced by using the proposed PDE on Spark. From the numerical experiment conducted on a variety of benchmark problems, it was confirmed that the speedup achieved by PDE generally increased as the number of computing partitions increased under certain range.

In our future work, we need to parallelize the three operators, i.e., mutation, crossover, and selection and the evaluation of objective function together. Besides, we would like to utilize PDE with more partitions to solve expensive problems such as CEC 2010 large scale benchmark problems [20] which need more than two hundred hours to finish the optimization task with single computer. And parallelizing other EAs with RDD is also very interesting work.

Acknowledgments. This work is partially supported by Natural Science Foundation of China under grant No. 61364025, State Key Laboratory of Software Engineering Foundation under grant No. SKLSE2012-09-39 and the Science and Technology Foundation of Jiangxi Province, China under grant No. GJJ13729 and No. GJJ14742.

References

1. Store, R., Price, K.V.: Differential evolution CA simple and efficient heuristic for global optimization over continuous spaces. *J. Glob. Optim.* **11**(4), 341–359 (1997)
2. Vesterstrom, J., Thomsen, R.: A comparative study of differential evolution, particle swarm optimization, and evolutionary algorithms on numerical benchmark problems, pp. 1980–1987 (2007)
3. Yousefi, H., Handroos, H., Soleymani, A.: Application of differential evolution in system identification of a servo-hydraulic system with a flexible load. *Mechatron.* **18**(9), 513–528 (2008)
4. Rocca, P., Oliveri, G., Massa, A.: Differential evolution as applied to electromagnetics. *Antennas Propag. Mag.* **53**(1), 38–49 (2011)
5. Wang, Y., Li, H.X., Huang, T.: Differential evolution based on covariance matrix learning and bimodal distribution parameter setting. *Appl. Softw. Comput.* **18**, 232–247 (2014)
6. Wang, Y., Cai, Z., Zhang, Q.: Enhancing the search ability of differential evolution through orthogonal crossover. *Inf. Sci.* **185**(1), 153–177 (2012)
7. Wang, Y., Cai, Z., Zhang, Q.: Differential evolution with composite trial vector generation strategies and control parameters. *IEEE Trans. Evol. Comput.* **15**(1), 55–66 (2011)
8. Alba, E., Tomassini, M.: Parallelism and evolutionary algorithms. *IEEE Trans. Evol. Comput.* **6**(5), 443–462 (2002)
9. Zaharie, D., Petcu, D.: Parallel implementation of multi-population differential evolution. *Concurrent Inf. Process. Comput.* **48**, 223–232 (2005)
10. Wang, H., Rahnamayan, S., Wu, Z.: Parallel differential evolution with self-adapting control parameters and generalized opposition-based learning for solving high-dimensional optimization problems. *J. Parallel Distrib. Comput.* **73**(1), 62–73 (2013)
11. Fabris, F., Krohling, R.A.: A co-evolutionary differential evolution algorithm for solving minCmax optimization problems implemented on GPU using C-CUDA. *Expert Syst. Appl.* **39**(12), 10324–10333 (2012)
12. Dean, J., Ghemawat, S.: MapReduce: Simplified data processing on large clusters. *Commun. ACM.* **51**(1), 107–113 (2008)
13. Zhou, C.: Fast parallelization of differential evolution algorithm using MapReduce. In: Proceedings of the 12th Annual Conference on Genetic and Evolutionary Computation, pp. 1113–1114, Dublin, Ireland (2011)
14. Pavlech, M.: Framework for development of distributed evolutionary algorithms based on MapReduce. In: Proceedings of the 22nd International DAAAM Symposium on Intelligent Manufacturing and Automation: Power of Knowledge and Creativity, pp. 1475–1476, Vienna (2011)
15. McNabb, A.W., Monson, C.K., Seppi, K.D.: Parallel PSO using mapreduce. In: Proceedings of IEEE Congress on Evolutionary Computation, pp. 7–14. IEEE, Singapore (2007)
16. Verma, A., Llorca, X., Goldberg, D.E.: Scaling genetic algorithms using mapreduce. In: Proceedings of the Ninth International Conference on Intelligent Systems Design and Applications, pp. 13–18. IEEE, Pisa (2009)
17. Zaharia, M., Chowdhury, M., Das, T.: Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing. In: The 9th USENIX Conference on Networked Systems Design and Implementation, 2012, pp. 1–16. USENIX Association, Berkeley (2012)

18. Isard, M., Buidu, M., Yu, Y., et al.: Dryad: distributed data-parallel programs from sequential building blocks. *ACM SIGOPS Operating Syst. Rev.* **41**(3), 59–72 (2007)
19. Kiyoharu, T., Takashi, I.: Concurrent differential evolution based on MapReduce. *Int. J. Comput.* **4**(4), 161–168 (2010)
20. Tang, K., Li, X., Suganthan, K.: Benchmark Functions for the CEC'2010 Special Session and Competition on Large Scale Global Optimization. Technical report, IEEE (2009)