Scalable Distributed Genetic Algorithm Using Apache Spark (S-GA)

Fahad Maqbool[1(✉)], Saad Razzaq[1], Jens Lehmann[2,3], and Hajira Jabeen[2]

[1] University of Sargodha, Sargodha, Pakistan
{fahad.maqbool, saad.razzaq}@uos.edu.pk
[2] Bonn University, Bonn, Germany
{lehmann, jabeen}@cs.uni-bonn.de
[3] Fraunhofer IAIS, Sankt Augustin, Germany

Abstract. In this era of big data with facilities for advanced real-time data acquisition, the solutions to large-scale optimization problems are strongly desired. Genetic Algorithms are efficient optimization algorithms that have been successfully applied to solve a multitude of complex problems. The growing need for large-scale optimization, and inherent parallel evolutionary nature of the algorithms calls for new solutions exploiting existing parallel, in-memory, distributed computing frameworks like Apache Spark. In this paper, we present an algorithm for Scalable Genetic Algorithms using Apache Spark (S-GA). S-GA makes liberal use of rich APIs offered by Spark. We have tested S-GA on several numerical benchmark problems for large-scale continuous optimization containing up to 3000 dimensions, 3000 population size, and one billion generations. S-GA presents a variant of island model and minimizes the materialization and shuffles in RDDs for minimal and efficient network communication. At the same time it maintains the population diversity by broadcasting the best solutions across partitions after specified Migration Interval. We have tested and compared S-GA with the canonical Sequential Genetic Algorithm (SeqGA). S-GA has been found to be more scalable and it can scale up to large dimensional optimization problems while yielding comparable results.

Keywords: Apache Spark · Parallel genetic algorithms · Function optimization · Hadoop Map Reduce

1 Introduction

Owing to inherently decentralized nature of Genetic Algorithms (GA), a multitude of variants of Parallel GA (PGA) have been introduced in the literature [1, 2]. However, their application has remain limited to moderately sized optimization problems and the research focused mostly on speeding up the performance of otherwise time-consuming and inherently complex applications e.g. assignment and scheduling [11–13], or prediction [8], tasks. To deal with large-scale optimization problems multi-core systems and standalone clusters architectures have been proposed by Zheng et al. [6]. They have used distributed storage file system or distributed processing framework like Apache Hadoop, to achieve scalability in PGA [3–6]. Hadoop Map Reduce [7], is a

© Springer Nature Switzerland AG 2019
D.-S. Huang et al. (Eds.): ICIC 2019, LNCS 11643, pp. 424–435, 2019.
https://doi.org/10.1007/978-3-030-26763-6_41

reliable, scalable and fault tolerant framework for large scale computing. Hadoop requires writing data to HDFS after each iteration to achieve its resilience. In case of CPU bound iterative processing, e.g. in case of Genetic Algorithms, this I/O overhead is undesirable and substantially dominates the processing time. PGA has been explored for numerous interesting applications like software fault prediction [8], test suite generation [9], sensor placement [10], assignment and scheduling [11–13], dynamic optimization [23], adapting offspring population size and number of islands [24].

Researchers have made significant efforts to explore the intrinsically parallel nature of genetic algorithms using island model [16], and other PGA models [18]. A lot of efforts have also been made by implementing and testing these models on Hadoop framework [8, 17, 18, 25], by using map reduce strategy to improve scalability. PGAs have been implemented using distributed frameworks and the effectiveness is evaluated in terms of execution time, computation effort, solution quality in comparison with Sequential Genetic Algorithm (SeqGA). However, the above mentioned efforts have been tested on simple problems which have been solved using limited population size and small number of generations overlooking the scalability that can be achieved by using these frameworks to solve large-scale optimization problems. Apache Spark [14] is an open source distributed cluster computing framework that has gained popularity in recent years. It has been shown to be faster than Hadoop for large scale optimization problems. it especially works better for iterative processing [14]. Contrary to Hadoop, Spark keeps data in memory and uses lineage graphs to achieve resilience and fault tolerance. This makes computing faster and eliminates the I/O overhead of read/write to the Hadoop distributed file system (HDFS) incurred in case of map-reduce. Spark provides APIs for generic processing in addition to specialized libraries for SQL like operations [29], stream processing using concepts of mini-batches [26], iterative machine learning algorithms [30], and a Graph processing library [15]. Spark's efficient data processing has proven to be 100 times faster for in-memory operations and 10 times faster for disk operations when compared to Hadoop MapReduce [25].

In this paper, we propose a Scalable GA (S-GA) for large-scale optimization problems using Apache Spark. S-GA aims to reduce the communication overhead of Apache Spark by optimal resource utilization for large scale optimization tasks. This is contrary to the traditional island model [16], of PGA, where communication among different subpopulation islands is directly proportional to the population and solution size. In S-GA, the communication is independent of the population size and is limited by the *Migration Rate* and *Migration Interval*. Hence, reducing a significant amount of data transfer between parallel computations making it scalable and applicable to large scale problems. We have compared S-GA with SeqGA for continuous numerical optimization problems. The experiments have been performed on five different large scale benchmark problems. The results of S-GA have been compared with GA and found to be more efficient.

The paper is structured as follows: In Sect. 2, related work is discussed. SeqGA and proposed S-GA is explained in Sects. 3 and 4 respectively. Experiments and evaluations are discussed in Sect. 5. Finally, we discuss the Conclusions and future work in Sect. 6.

2 Related Work

Generally, there are three main models to parallelize GA i.e. global single-population master-slave (global model), single-population fine-grained (grid model), multiple-population coarse-grained (island model) [1]. Mostly, PGA divides a population into multiple sub-populations. Each population independently searches for an optimal solution using stochastic search operators like crossover and mutation. The **Global Model** works like SeqGA with one population. The master is responsible for handling the population by applying GA operators while slave manages the fitness evaluation of individuals. In **Grid Model**, GA operators are applied within each sub-population and each individual is assigned to only one sub-population. This helps in improving the diversity. However, this model suffers from the problem of getting stuck in a local optima, and it has high communication overhead due to frequent communication between the sub-populations. **Island Model**, [16] uses a large population divided among different sub-populations called islands. GA operates on these islands independently with the ability to exchange/migrate some of the individuals. This helps in increasing the diversity of chromosome and avoid to get stuck in a local optima. **SeqGA** uses single large population pool and apply stochastic operators on them. Details about SeqGA, is given in Sect. 3.2. Whitley et al. [16], expected that Island model would outperform SeqGA, because of the diversity of chromosomes and migration of individuals among several islands. However, results revealed that Island model may perform better only if migration among sub-populations is handled carefully.

A comparison of Hadoop Map Reduce based implementation of three main PGA models, global single-population master-slave GAs (global model), single-population fine-grained (grid model), multiple-population coarse-grained (island model) is discussed by Ferrucci et al. [8]. They observed that overhead of Hadoop distributed file system (HDFS) make Global and Grid models less efficient as compared to Island model for parallelizing GA for because of HDFS access, communication and execution time. Island model performs less HDFS operations, resulting in optimized resource utilization and efficient execution time. However, they reported experimental results of Global, Grid, and Island models on population size of 200 only, with 300 generations on smaller problems with a limited number of dimensions (only up to 18). Their results concluded that distributed frameworks provide efficient support for data Distribution, parallel processing, and memory management but they incur significant overhead of communication delays.

Verma et al. [17], used Hadoop MapReduce framework to make GA scalable. Their experiments were performed on OneMax problem and they addressed the scalability and convergence as decreasing time per iteration, by increasing the number of resources while keeping the problem size fixed. Keco and Subasi [18] discussed PGA using Hadoop MapReduce. Their focus was to improve final solution quality and cloud resource utilization. They obtained improved performance and fast convergence but there were no improvements in the solution quality due to lack of communication among the subpopulations. Edgar et al. [19], proposed a diversity based parent selection mechanism for speeding up the multi-objective optimization using Evolutionary Algorithm. This novel parent selection mechanism helped to find the Pareto front faster

than the classical approaches. Osuna et al. [19], focused on individuals having high diversity located in poor explored areas of the search space. Gao et al. [20], contributed to maximizing the diversity of population in GA, by favoring the solutions whose fitness value is better than a given threshold. They worked on OneMax and Leading One's [27], problems. The results revealed that algorithm efficiently maximized the diversity of a population. They have presented a theoretical framework and haven't addressed the contribution of diversity in large-scale optimization problems.

PGA using Apache Spark framework [9], was proposed for the pairwise test suite generation. Parallel operations were used for fitness evaluation and genetic operations. They did not address the large scale data problems and only focused on test suite size generation. Results were compared with SeqGA on synthetic and real-world datasets [9].

Both, GA and PGAs are widely used in several applications. Junior et al. [21], applied parallel biased random-key GA with multiple populations on irregular strip packing problem. In this problem, items of variable length and fixed width need to be placed in a container. For an efficient layout scheme, they used collision-free region as a partition method along with a meta-heuristic and a placement algorithm. Gronwald et al. [22], determined location and amount of pollutant source in air by using Backward Parallel Genetic Algorithm (BPGA). A concentration profile was compiled by considering the readings from different points in an area. BPGA utilized multiple guesses in a generation, and the best one was determined by a fitness function. This best guess was used in the reproduction of next generation.

Previously proposed parallel implementations of GA majorly differ in structuring the population and subpopulations named as the topology. The topology of PGA determines the sub-population model and the sharing of solutions (i.e. sending and receiving solutions from each other) among these subpopulations. These models, when executed using distributed frameworks like Apache Spark, suffer from substantial communication and network overhead. On one hand there is substantial parallelism intrinsic in Genetic Algorithms, and on the other hand, the desired communication hinders the ideal speed-up that could be achieved by using parallel/distributed techniques. There exists a tradeoff between sub-population communication and solution quality (due to population stagnation, getting stuck in the local optima and the lack of diversity).

In conclusion, there is a strong need to develop fundamental new approaches for parallel and distributed GA, while keeping in view the I/O, network, and communication overhead present in the existing distributed large scale computing frameworks. In order to exploit the existing frameworks, the implementation must make optimal resource utilization to gain the ideal speedup.

3 Background

3.1 Apache Spark

Apache Spark [28] was introduced by RAD Lab at the University of California in 2009 in order to overcome the limitations of Hadoop MapReduce. It has been designed for faster in-memory computation for interactive queries and iterative algorithms while

achieving efficient fault recovery and compliance with the Hadoop stack. At its core, Spark is a "computational engine" that is responsible for scheduling, distributing, and monitoring applications consisting of many computational tasks across many worker machines, or a computing cluster. Apache Spark provides data distribution using resilient distributed datasets (RDDs), which are Spark's main programming abstraction. RDDs represent a collection of items distributed across many compute nodes that can be manipulated in parallel. RDD supports two types of operations: (i) Transformations, (ii) Actions. Transformations are lazy operations that create a new RDD from existing data in RDD. Lazy evaluation means that transformations are not executed, and an execution graph is created instead, until an action is called. The actions materialize the lazy evaluations and perform operations (e.g. aggregation) that transfer data from worker nodes to the master node. In order to efficiently work with RDDs it is important to be aware of the internal working details of RDDs, use of narrow transformations and dependencies, reducing the number of actions etc. in order to achieve better speed up with the parallel computing.

3.2 Sequential Genetic Algorithm (SeqGA)

SeqGA [11–13, 16, 18, 20], also known as Canonical GA is a stochastic search method that is used to find the optimal solution for a given optimization problem using the Darwinian's principal of evolution "Survival of the Fittest". It creates a single pool of possible solutions population (panmixia) and applies stochastic operators (i.e. Selection, Crossover, Mutation, and Survival Selection) to create a new evolved population. This process of new population evolution continues until the population has converged to an optimal solution, or desired time/effort has elapsed. For large scale, or complex problems, SeqGA may require more computational effort like more memory and long execution time (for large population size and more generations).

Algorithm 1 explains the working of SeqGA. (Line 3), *Select Parents* specifies the individual selection mechanism for reproduction or recombination. *Crossover* (Line 4) helps to explore the search space by generating new solutions after recombination, while M*utation* (Line 5) exploits the solutions for improvement by random perturbation of the selected solution. The *Survival Selection* scheme decides the number of individuals to be selected from parents and offspring's for the next generation.

Algorithm 1. Sequential Genetic Algorithm

1.P ← *Generate Initial Population*

2.**While** Stopping Criteria not met **do**

3. *P'← Select Parents (P)*

4. *P' ← Crossover (P')*

5. P' ← *Mutate (P')*

6. P' ← *Survival –Selection (P U P')*

7. P ← P'

end while

4 Scalable Distributed Genetic Algorithm Using Apache Spark (S-GA)

S-GA creates an initial random population of solutions and distributes them on different partitions as an RDD. The GA operators and fitness evaluations are performed within each partition, independent of the other partitions. We have used roulette wheel selection operator, uniform crossover, interchange mutation operation, and weak parent survival selection, for creation of new offspring's for the next generation.

In S-GA each partition (corresponding to an island in island model) replaces its weakest solution by the fittest solutions broadcasted by other partitions. **Migration Size** *(Ms)* specifies the number of solutions to be broadcasted to other partitions during each migration step. S-GA significantly reduces the communication overhead by minimizing the actions on RDD.

The pseudo code of S-GA is elaborated in Algorithm 2. The population is randomly initialized at line (1) then distributed among *m* partitions at line (2). Solutions are evolved using stochastic operators at line (6–12). It is worth mentioning here that we have used operations that calculate and sort the fitness within each partition (MapPartitionsWithIndex), therefore reducing the communication overhead and achieving efficient performance. At line (14), SGA broadcasts evolved best solutions (s) to other partitions and the weak solutions from the partitions are replaced with the new broadcasted solutions at line (6). Migration Interval (*Mi*) defines the number of generations after which S-GA broadcasts the fittest individual (s) of each partition to other partitions. This helps achieving diversity in each subpopulation while searching for the better solutions. The size of the broadcast and Migration Interval contribute to the network communication delay, and directly affect the performance and convergence. We have experimented with several values in our evaluations. Finally the above steps are iterated until the stopping criteria is met. Figure 1 explains the idea of Migration, Migration Size and Migration Interval with an example.

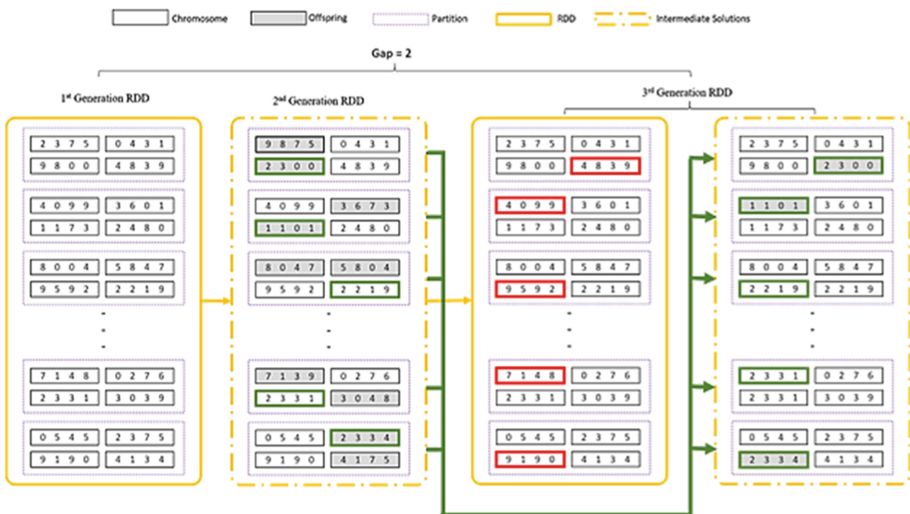

Fig. 1. Evolution process of S-GA

Let's assume value of $Mi = 2$, $Ms = 1$, and fitness function as sphere (i.e., $f(xi) = \sum_{i=1}^{n} x_i^2$. Initial RDD is created using a population of random solutions. These initial solutions are then evolved using crossover and mutation operators. After every 2nd generation (as $Mi = 2$), best solution (as $Ms = 1$) from each partition is migrated to other partitions. As the solution migrates, each partition at the start of very next generation picks all the migrated solutions and replaces them with its weakest solutions at each partition.

Algorithm 2. Pseudo-code-of-S-GA

N: Population Size
P: Population
P_i: Sub-Population at partition i
D: Dimensions
G: Generations
m: Number of Partitions
M_i: Migration Interval / gap
f: Fitness Function
M_s: Migration Size

1: Randomly initialise population of size P
2: Distribute P among m partitions
3: G = 0+
4: **while** stopping criteria not met **do**
5: at each partition i
6: **for** k: 1 to M_i **do**
7: $P_i'' \leftarrow$ Select Parents (P_i')
8: $P_i'' \leftarrow$ *Crossover* (P_i')
9: $P_i'' \leftarrow$ *Mutate* (P_i')
10 *Calculate Fitness* (P_i'')
11: $P_i' \leftarrow$ *Survival_ Selection* ($P_i' \cup P_i''$)
12: **end for**
13: BroadcastSolutions
13: End at each partition i
14. $P_{i'}$ =(P_i - (weak (m* M_s) solutions)) ∪ BroadcastSolutions
15: G = G + M_i
16: **end while**

5 Experiments

5.1 Experimental Setup

The experiments are performed on a three node cluster: DELL PowerEdge R815, 2x AMD Opteron 6376 (64 Cores), 256 GB .RAM, 3 TB SATA RAID-5 with spark-2.1.0 and Scala 2.11.8. Both S-GA and SeqGA used Crossover scheme: Uniform, Mutation: Interchange, Replacement Scheme: Weak parent, Selection Scheme: Roulette Wheel, Crossover Probability: 0.5, Mutation Probability: 0.05, P = D, and Function: Griewank as configuration parameters. While S-GA also used m: 24 and M_s: 2 as configuration parameters.

5.2 Evaluation Matrics

Speed Up: It is the ratio of sequential execution time to the parallel execution time. It reflects how much parallel algorithm is faster than a sequential algorithm. Table 1 reflects speed up for all the cases where SeqGA and S-GA converge to VTR (Value To Reach). VTR defines the threshold for convergence. We have used $\frac{1}{Number\ of\ Dimensions}$ as VTR in experimentations.

In Table 1, we with different values of Migration Interval and Migration Size. it can be seen that for large Migration Interval and Migration Size, a high speedup was achieved.

Execution Time: The execution time of SeqGA and S-GA was measured using system clock time. This time was recorded for a maximum of 1 billion generations. Table 2 shows average execution time over 5 runs for each configuration of S-GA. We can observe that execution time reduces significantly when we increase M_i from 50000 to 100000, however fitness error also decreases significantly. This difference in time reduces with an increase in the number of partitions. Migration overhead defines the total number of migrated individual (s) by all partitions after M_i. Increase in m and M_i results in increased network overhead (m* M_i) and hence execution time. But on the other hand this also helps S-GA to converge in a lesser number of generations. Table 2 lists the execution time of Sphere, Ackley, Griewank, Rastrigin, Zakharov, and Sum-of-Different-Power-functions for optimization upto 3000 dimensions (D). For simplicity population size (N) has been assumed to be equivalent to the number of dimensions. G represents the number of generations that have been consumed using specified configurations. VTR as mentioned earlier, is reciprocal to D. Hence VTR would be lesser for 3000 dimensions compared to 2000 and 1000 dimensions. Bold values in Table 2 represents the fitness error that has decreased beyond the specified threshold i.e. VTR.

Table 1. Experimental results of S-GA and SeqGA.

D	SeqGA						S-GA			Speed up
	G	Time	Error	M_i	m	M_s	G	Time	Error	
1000	748	2476999	2.45e-4	25000	18	1	106850000	712	8.25e-4	–
						2	39050000	352	9.28e-4	–
						3	27275000	327	1.4e-4	–
					24	1	46050000	356	6.68e-4	–
						2	22675000	263	8.08e-4	–
						3	19925000	311	9.55e-4	–
					30	1	37500000	328	2.87e-4	–
						2	15750000	228	2.33e-4	1.08
						3	13150000	269	7.01e-4	–
				50000	18	1	194500000	650	2.79e-4	–
						2	86500000	393	7.99e-4	–
						3	54850000	317	4.15e-4	–
					24	1	92800000	565	9.37e-5	–
						2	44250000	262	2.52e-4	–
						3	38750000	311	3.69e-4	–
					30	1	81850000	344	3.42e-4	–
						2	33550000	244	2.01e-5	1.01
						3	30350000	309	3.24e-4	–
2000	989	1064	2.03e-4	25000	18	1	242650000	3479	1.94e-4	–
						2	95400000	1845	4.17e-4	–
						3	62225000	1472	1.67e-4	–
					24	1	127075000	2112	2.65e-4	–
						2	61875000	1466	3.22e-4	–
						3	36650000	1061	3.51e-4	1.002
					30	1	95950000	1714	1.88e-4	–
						2	41250000	1042	2.64e-4	1.02
						3	26275000	970	1.65e-4	1.1
				50000	18	1	448200000	3309	1.5e-4	–
						2	179300000	1713	2.37e-4	–
						3	133200000	1541	4.15e-5	–
					24	1	246500000	2162	3.62e-4	–
						2	120250000	1429	2.47e-4	–
						3	74300000	1052	1.37e-4	1.01
					30	1	185250000	1837	2.91e-4	–
						2	78900000	1076	3.2e-4	–
						3	54900000	989	2.56e-4	1.07

Table 2. Experimental Results of S-GA.

f	Mi	D = 1000			D = 2000			D = 3000		
		VTR = 0.001			VTR = 5.0 E-4			VTR = 3.33 E-4		
		G	Time	Error	G	Time	Error	G	Time	Error
Sphere	50000	1e9	556	8.28	1e9	11531	0.004	1e9	18904	0.017
	100000	1e9	282	265.05	1e9	5932	0.003	1e9	12227	5.967
Ackley	50000	1e9	5616	0.009	1e9	11799	0.095	1e9	17609	1.27
	100000	1e9	2613	0.02	1e9	5819	0.015	1e9	9456	2.07
Griewank	50000	4.4e7	262	**2.52e-4**	1.2e8	1429	**2.47e-4**	2.1e9	3917	**1.25e-4**
	100000	9.6e7	277	**6.23e-4**	2.4e6	1417	**1.41e-4**	4.1e9	3732	**2.47e-4**
Rastrigin	50000	1e9	5339	0.024	1e9	11513	1.443	1e9	18594	0.067
	100000	1e9	2623	2.081	1e9	5809	0.907	1e9	9447	52.45
Zakharov	50000	1e9	5575	17035.15	1e9	11779	33111.93	1e9	16048	10249.73
	100000	1e9	2896	16803.21	1e9	5783	33205.55	1e9	9036	50674.89
Sum of Diff Powers	50000	200000	6	**4.59e-4**	250000	10	**3.16e-4**	700000	19	**1.6e-4**
	100000	400000	6	**2.82e-4**	400000	8	**4.5e-4**	600000	11	**1.42e-4**

It can be seen from Table 2 that for higher values of M_i (i.e. 100000), each function consumes less time in most of the cases. Broadcasts are also important as they help each sub-population P_j to increase it's diversity and helps each P_j to get out of local optima. Increased M_i values reduces frequent broadcasts and hence the network overhead. In case of higher M_i, more number of iterations may not improve the optima significantly, due to reduced diversity in the particular sub population. Table 2 reveals the discussed fact as Error is less for M_i = 50000 as compared to M_i= 100000 in most of the cases.

6 Conclusion

In this paper, we have proposed initial results for S-GA using Apache Spark for large-scale optimization problems. The results have been compared with SeqGA. We have tested S-GA for Sphere, Ackley, Griewank, Rastrigin, Zakharov, and Sum-of-Different-Powers functions that are typical benchmarks for continuous optimization problems. We have used population size of up to 3000, Dimensions of up to 3000, Partition Size up to 30, Migration Size up to 03, and Migration Interval to 100000. For few cases S-GA has outperformed SeqGA for higher Population, Partitions, Migration Size, and Migration Interval in term of execution time. In future, we plan to extend S-GA and evaluate different migration and distribution strategies for larger scale and more complex optimization problems.

Acknowledgment. This work was partly supported by the EU Horizon2020 projects Boost4.0 (GA no. ~ 780732), LAMBDA (GA no. ~ 809965), SLIPO (GA no. ~ 731581), and QROWD (GA no. ~ 723088).

References

1. Luque, G., Alba, E.: Parallel Genetic Algorithms: Theory and Real-World Applications. Springer, Heidelberg (2011)
2. Knysh, D.S., Kureichik, V.M.: Parallel genetic algorithms: a survey and problem state. J. Comput. Syst. Sci. Int. **49**(4), 579–589 (2010)
3. Chávez, F., et al.: ECJ + HADOOP: an easy way to deploy massive runs of evolutionary algorithms. In: Squillero, G., Burelli, P. (eds.) EvoApplications 2016. LNCS, vol. 9598, pp. 91–106. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-31153-1_7
4. Di Geronimo, L., Ferrucci, F., Murolo, A., Sarro, F.: A parallel genetic algorithm based on hadoop MapReduce for the automatic generation of JUnit test suites: In: IEEE International Conference on Software Testing, Verification and Validation (2012)
5. Salza, P., Ferrucci, F., Sarro, F.: Develop, deploy and execute parallel genetic algorithms in the cloud. In: Genetic and Evolutionary Computation Conference (GECCO) (2016)
6. Zheng, L., Lu, Y., Ding, M., Shen, Y., Guoz, M.: Architecture-based performance evaluation of genetic algorithms on multi/many-core systems. In: IEEE International Conference on Computational Science and Engineering (2011)
7. Hashem, I.T., Anuar, N.B., Gani, A.Y., Xia, F., Khan, S.U.: MapReduce review and open challenges. Scientometrics **109**, 389–422 (2016)
8. Ferrucci, F., Pasquale, S., Federica, S.: Using hadoop MapReduce for parallel genetic algorithm: a comparison of the global, grid and island models. Evol. Comput. Early Access **26**(4), 535–567 (2017)
9. Qi, R.Z., Wang, Z.J., Li, S.-Y.: A parallel genetic algorithm based on spark for pairwise test suite. J. Comput. Sci. Technol. **31**(2), 417–427 (2016)
10. Hu, C., Ren, G., Liu, C., Li, M., Jie, W.: A spark-based genetic algorithm for sensor placement in large-scale drinking water distribution systems. Cluster Comput. J. Netw. Softw. Tools Appl. **20**(2), 1089–1099 (2017)
11. Lim, D., Ong, Y.-S., Jin, Y., Sendhoff, B., Lee, B.-S.: Efficient hierarchical parallel genetic algorithm using grid computing. Future Gener. Comput. Syst. **23**(4), 658–670 (2007)
12. Liu, Y.Y., Wang, S.: A scalable parallel genetic algorithm for the generalized assignment problem. Parallel Comput. **46**, 98–119 (2015)
13. Trivedi, A., Srinivasan, D., Biswas, S., Reindl, T.: Hybridizing genetic algorithm with differential evolution for solving the unit commitment scheduling problem. Swarm Evol. Comput. **23**, 50–64 (2015)
14. Gu, L., Li, H.: Memory or time performance evaluation for iterative operation on hadoop and spark. In: High-Performance Computing and Communications and IEEE International Conference on Embedded and Ubiquitous Computing (HPCC EUC) (2013)
15. Wani, M.A., Jabin, S.: Big data: issues, challenges, and techniques in business intelligence. In: Aggarwal, V.B., Bhatnagar, V., Mishra, D.K. (eds.) Big Data Analytics. AISC, vol. 654, pp. 613–628. Springer, Singapore (2018). https://doi.org/10.1007/978-981-10-6620-7_59
16. Whitley, D., Rana, S., Heckendorn, R.B.: The island model genetic algorithm: on separability, population size, and convergence. CIT J. Comput. Inf. Technol. **7**(1), 33–47 (1999)
17. Verma, A., Llorà, X., Goldberg, D.E., Campbell, R.H.: Scaling simple, compact and extended compact genetic algorithms using MapReduce. Illinois Genetic Algorithms Laboratory (Illinois) report no. 2009001, illegal, University of Illinois, Urbana-Champaign (2009)
18. Keˇco, D., Subasi, A.: Parallelization of genetic algorithms using hadoop Map/Reduce. SouthEast Eur. J. Soft Comput. **1**(2), 56–59 (2002)

19. Osuna, E.C., Gao, W., Neumann, F., Sudholt, D.: Speeding up evolutionary multi-objective optimization through diversity-based parent selection. In: Genetic and Evolutionary Computation Conference, Berlin, Germany (2017)
20. Gao, W., Neumann, F.: Runtime analysis of maximizing population diversity in single-objective optimization. In: Genetic and Evolutionary Computation Conference, Vancouver, Canada (2014
21. Junior, B.A., Pinheiro, P.R., Coelho, P.V.: A parallel biased random-key genetic algorithm with multiple populations applied to irregular strip packing problems. Math. Probl. Eng. **2017**, 11 (2017)
22. Gronwald, F., Chang, S., Jin, A.: Determining a source in air dispersion with a parallel genetic algorithm. Int. J. Emerg. Technol. Adv. Eng. **7**(8), 174–185 (2017)
23. Lissoni, A., Witt, C.: A runtime analysis of parallel evolutionary algorithms in dynamic optimization. Algorithmica **78**(2), 641–659 (2017)
24. Lässig, J., Sudholt, D.: Adaptive population models for offspring populations and parallel evolutionary algorithms. In: 11th Workshop Proceedings on Foundations of Genetic Algorithms, Schwarzenberg, Austria (2011)
25. Shoro, A.G., Soomro, T.R.: Big data analysis: apache spark perspective. Global J. Comput. Sci. Technol. **15**(1), 09–14 (2015)
26. Zaharia, M., et al.: Apache spark: a unified engine for big data processing. Commun. ACM **59**(11), 56–65 (2016)
27. Witt, C.: Runtime analysis of the ($\mu + 1$) EA on simple pseudo-Boolean functions. Evol. Comput. **14**(1), 65–86 (2006)
28. Zaharia, M., et al.: Apache spark: a unified engine for big data processing. Commun. ACM **59**(11), 59–65 (2016)
29. Armbrust, M., 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, May 2015
30. Meng, X., et al.: MLlib: machine learning in apache spark. J. Mach. Learn. Res. **17**(1), 1235–1241 (2016)