Implementing Parallel Differential Evolution on Spark

Diego Teijeiro[1], Xoán C. Pardo[1], Patricia González[1], Julio R. Banga[2], and Ramón Doallo[1]

[1] Grupo de Arquitectura de Computadores. Universidade da Coruña. Spain ({diego.teijeiro,xoan.pardo,patricia.gonzalez,doallo}@udc.es)
[2] BioProcess Engineering Group. IIM-CSIC. Spain (julio@iim.csic.es)

Abstract. Metaheuristics are gaining increased attention as an efficient way of solving hard global optimization problems. Differential Evolution (DE) is one of the most popular algorithms in that class. However, its application to realistic problems results in excessive computation times. Therefore, several parallel DE schemes have been proposed, most of them focused on traditional parallel programming interfaces and infrastructures. However, with the emergence of Cloud Computing, new programming models, like Spark, have appeared to suit with large-scale data processing on clouds. In this paper we investigate the applicability of Spark to develop parallel DE schemes to be executed in a distributed environment. Both the master-slave and the island-based DE schemes usually found in the literature have been implemented using Spark. The speedup and efficiency of all the implementations were evaluated on the Amazon Web Services (AWS) public cloud, concluding that the island-based solution is the best suited to the distributed nature of Spark. It achieves a good speedup versus the serial implementation, and shows a decent scalability when the number of nodes grows.
Keywords: Metaheuristics, Differential Evolution, Cloud Computing, Spark, Amazon Web Services

1 Introduction

Global optimization problems arise in many areas of science and engineering [1–3]. Most of these problems are NP-hard, so many research efforts have focused on developing metaheuristic methods which are able to locate the vicinity of the global solution in reasonable computation times. Moreover, in order to reduce the computational cost of these methods, a number of researchers have studied parallel strategies for metaheuristics [4, 5]. However, all these efforts are focused on traditional parallel programming interfaces and traditional parallel infrastructures.

With the advent of Cloud Computing effortless access to large number of distributed resources has become more feasible. But developing applications that execute at so big scale is hard. New programming models are being proposed to deal with large scale computations on commodity clusters and Cloud resources.

Distributed frameworks like MapReduce [6] or Spark [7] provide high-level programming abstractions that simplify the development of distributed applications including implicit support for deployment, data distribution, parallel processing and run-time features like fault tolerance or load balancing. We wonder how much benefit can we expect from implementing parallel metaheuristics using these new programming models because, besides the many advantages, they also have some shortcomings. Cloud-based distributed frameworks prefer availability to efficiency, being the speedup and distributed efficiency frequently lower than in traditional parallel frameworks due to the underlying multitenancy of virtualized resources.

The aim of this paper is to explore this direction further considering a parallel implementation of Differential Evolution (DE) [8], probably one of the most popular heuristics for global optimization, to be executed in the Cloud. The main contribution of the proposal is an analysis of different alternatives in implementing parallel versions of the DE algorithm using Spark and a thoroughly evaluation of their feasibility to be executed in the Cloud using a real testbed on the Amazon Web Services (AWS) public cloud.

The organization of this paper is as follows. Section 2 briefly presents the background and related work. Some new programing models in the Cloud are described in Section 3. Section 4 describes the proposed implementations of the Differential Evolution algorithm using Spark. The performance of the proposal is evaluated in Section 5. Finally, Section 6 concludes the paper and discusses future work.

2 Related Work

The parallelization of metaheuristics methods has received much attention to reduce the run time for solving large-scale problems [9]. Many parallel algorithms have been proposed in the literature, most of them being parallel implementations based on traditional parallel programming interfaces such as MPI and OpenMP. However, research on cloud-oriented parallel metaheuristics based mainly on the use of MapReduce has also received increasing attention in recent years. MRPSO [10] uses the MapReduce model to parallelize the Particle Swarm Optimization (PSO). MRPGA [11] attempts at combining MapReduce and genetic algorithms (GA). They properly claim that GAs cannot be directly expressed in MapReduce due to their specific characteristics. So they extend the model featuring a hierarchical reduction phase. A different approach is followed in [12], that tries to hammer the GAs into the MapReduce model. In [13] the applicability of MapReduce to distributed simulated annealing (SA) was also investigated. They design different algorithmic patterns of distributed SA with MapReduce and evaluate their proposal on the AWS public cloud. Recently, in [14], a practical framework to infer large gene networks through a parallel hybrid GA-PSO optimization method using MapReduce has also been proposed.

Some proposals are more specific on studying how to apply MapReduce to parallelize the DE algorithm to be used in the Cloud. In [15] the fitness evaluation in the DE algorithm is performed in parallel using Hadoop (the well-known open-source MapReduce framework). However, the experimental results reveal that the extra cost of Hadoop DFS I/O operations and the system bookkeeping overhead significantly reduces the benefits of the parallelization. In [16], a concurrent implementation of the DE based on MapReduce is proposed, however, it is a parallelized version of a neoteric DE based on the steady-state model instead of on the generation alternation model. While the generational model holds two populations and generates all individuals for the second population from those of the current population, the steady-state model holds only one population and each individual of the population is updated one by one. Comparing with the generational model, the parallelization of the steady-state model is simpler because it does not require synchronization for replacing the current population by newborn individuals simultaneously. On the other hand, the experiments reported in that paper were conducted on a multi-core CPU, thus, their implementation take advantage of the shared-memory architecture, sharing the population among the different threads, which is not possible in a distributed cloud environment. In [17] a parallel implementation of DE based clustering using MapReduce is also proposed. This algorithm was implemented in three levels, each of which consists of different DE operations.

To the best of our knowledge, there is no previous work that explores the use of Spark for evolutionary computation. Also, previous works using MapReduce have rarely evaluated their proposals in a real testbed on a public cloud.

3 New Programming Models in the Cloud

From the new programming models that have been proposed to deal with large scale computations on cloud systems, MapReduce [6] is the one that has attracted more attention since its appearance in 2004. In short, MapReduce executes in parallel several instances of a pair of user-provided *map* and *reduce* functions over a distributed network of *worker* processes driven by a single *master*. Executions in MapReduce are made in batches, using a distributed filesystem (typically HDFS) to take the input and store the output. MapReduce has been applied to a wide range of applications, including distributed pattern-based searching, distributed sorting, graph processing, document clustering or statistical machine translation among others.

When it comes to iterative algorithms as those that are typical in areas like machine learning or evolutionary computation, MapReduce has shown serious performance bottlenecks. Computations in MapReduce can be described as a *directed acyclic data flow* where a network of stateless mappers and reducers process data in single batches (see Figure 1). All input, output and intermediate data is stored and accessed via the file system and map/reduce tasks are created in every single batch. Having several of these single batches executed inside

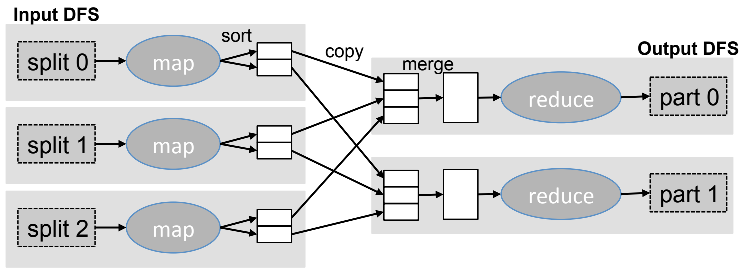


Fig. 1: MapReduce dataflow.

a loop has shown to introduce considerable performance overhead [18] mainly because there is no way of reusing data or computation from previous iterations efficiently.

Although some extensions have been proposed to improve the support to iterative algorithms in MapReduce like Twister [18], iMapReduce [19], or HaLoop [20], they still perform poorly on the kind of algorithms we are interested in, mainly due to those systems inability to exploit the (sparse) computational dependencies present in these tasks [21]. New proposals, not based on MapReduce, like Spark [7] or Fink (which has its roots on Stratosphere [22]), are designed from the very beginning to provide efficient support for iterative algorithms.

Spark provides a language-integrated programming interface to *resilient distributed datasets* (RDDs), a distributed memory abstraction for supporting fault-tolerant and efficient in-memory computations. According to authors [7] the performance of iterative algorithms can be improved by an order of magnitude when compared to MapReduce (using Hadoop).

Formally, an RDD is a read–only fault–tolerant partitioned collection of records. Users can manipulate them using a rich set of operators, control their partitioning to optimize data placement and explicitly persist intermediate results (in memory by default but also to disk). RDDs are created from other RDDs or from data in stable storage by applying coarse-grained *transformations* (e.g., *map*, *filter* or *join*) that apply the same operation to many data items. Once created, RDDs are used in *actions* (e.g. *count*, *collect* or *save*) which are operations that return a value to the application or export data to a storage system.

RDDs are computed lazily the first time they are used in an action, so transformations can be pipelined to form a *lineage*. By storing enough information about their lineages, RDDs do not need to be materialized at all times, as every RDD can recompute its partitions from previously persisted RDDs or data in stable storage at any time. This feature is the one used to provide fault-tolerance in case of an RDD partition lost.

Spark runtime is composed of a single *driver* program and multiple *workers* which are long-lived processes launched by the driver. Workers read data blocks from a distributed file system and persist RDD partitions in RAM across opera-

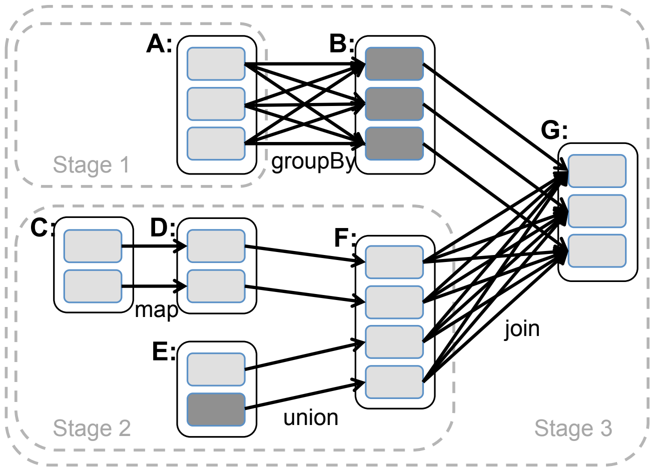


Fig. 2: Example of how Spark computes job stages.

tions. Developers write the driver program where they define one or more RDDs and invoke actions on them. Whenever an action is executed on an RDD, the Spark job scheduler uses its lineage to compute a *directed acyclic graph* (DAG) of stages. The scheduler then launches tasks to compute missing partitions from each stage until it has computed the target RDD. Assignment of tasks to workers takes into account data locality. Tasks end up being assigned to workers that already hold the RDD partitions of interest in memory.

An example of how Spark computes job stages is showed in Figure 2. In the figure, boxes with solid outlines are RDDs. Partitions are shaded rectangles, darker if they are persisted in memory. Each stage contains as many pipelined transformations with *narrow* (one-to-one or one-to-many) dependencies as possible. The boundaries of the stages (boxes with doted outlines in the figure) are the shuffle operations required for *wide* (many-to-one or many-to-many) dependencies or the presence of an already computed RDD in the lineage. In the example, to run an action on RDD G, as output RDD from stage 1 is already in RAM only stages 2 and 3 need to be executed.

4 Implementing Differential Evolution on Spark

Differential Evolution is an iterative mutation algorithm where vector differences are used to create new candidate solutions. Starting from an initial population matrix composed of NP D-dimensional solution vectors (individuals), DE attempts to achieve the optimal solution iteratively through changes in its vectors. Algorithm 1 shows the basic pseudocode for the DE algorithm. For each iteration, new individuals are generated in the population matrix through

Algorithm 1: Differential Evolution algorithm (`seqDE`)

```
input  : A population matrix $P$ with size $D$ x $NP$
output: A matrix $P$ whose individuals were optimized
repeat
    for each element $x$ of the $P$ matrix do
        $\overrightarrow{a}, \overrightarrow{b}, \overrightarrow{c} \leftarrow$ different random individuals from P matrix
        for $k \leftarrow 0$ to $D$ do
            if random point is less than $CR$ then
                $\overrightarrow{Ind}(k) \leftarrow \overrightarrow{a}(k) + F(\overrightarrow{b}(k) - \overrightarrow{c}(k))$
            end
        end
        if Evaluation($\overrightarrow{Ind}$) is better than Evaluation($\overrightarrow{P(x)}$) then
            Replace_Individual($P$,$\overrightarrow{Ind}$)
        end
    end
until Stop conditions;
```

operations performed among individuals of the matrix (mutation - F), with old solutions replaced (crossover - CR) only when the fitness value of the objective function is better than the current one.

A population matrix with optimized individuals is obtained as output of the algorithm. The best of these individuals are selected as solution close to optimal for the objective function of the model. However, in some real applications, such as parameter estimation in dynamic models, the performance of the classical sequential DE is not acceptable due to the large number of objective function evaluations needed. As a result, typical runtimes for realistic problems are in the range from hours to days. Parallelism can help improving both computational time and number of iterations for convergence. In the literature, different parallel models can be found [9], being the most popular ones the *master-slave* model and the *island-based* model. In the *master-slave* model the inner-loop of the algorithm is parallelized. A master processor distributes computation operations between the slave processors. Therefore, the parallel algorithm has the same behavior of the sequential one. In the *island-based* model the population matrix is divided in subpopulations (*islands*) where the algorithm is executed isolated. Sparse individual exchanges are performed among islands to introduce diversity into the subpopulations, preventing search from getting stuck in local optima.

With the aim of better understanding Spark intricacies and assess the performance of different alternatives when implementing DE, we have developed three different versions of the algorithm: (1) the classic sequential algorithm (`seqDE`) which has been implemented for comparative purposes and it is the only that does not make use of Spark; (2) three different variants of the master-slave parallel implementation (`SmsPDE`); and (3) an island-based parallel implementation (`SiPDE`). All of them have been coded using the Scala language [23] which is the one used to implement Spark itself although APIs for Python and Java also exist. The rest of this section features relevant facts about each of the implementations. As it will be demonstrated, the main conclusion is that the island-based parallel implementation is the best suited to the distributed nature of Spark and obtains the best performance results.

4.1 Master-Slave DE

To implement a master-slave parallel version of the DE algorithm using Spark, some previous insight into the way data is distributed and processed by Spark is needed. Spark uses the RDD abstraction to represent fault-tolerant distributed data. RDDs are inmutable sets of records that optionally can be in the form of key-value pairs. Spark driver (the master in Spark terminology) partitions RDDs and distributes the partitions to workers (the slaves in Spark terminology), which persist and transform them and return results to the driver. There is no communication among workers. Shuffle operations (i.e. join, groupBy) that need data movement among workers through the network are expensive and should be avoided.

Our Spark-based master-slave DE implementation (`SmsPDE`) follows the scheme shown in Figure 3. A key-value pair RDD has been used to represent the population where each individual is uniquely identified by its key. Some DE algorithm steps have been selected as appropriate to be executed in a distributed fashion:

- The random generation and initial evaluation of individuals that form the population, implemented as a Spark map transformation.
- The mutation strategy including random pick of individuals and replacement of old individuals with new improved ones, implemented using three different variants that are explained later.
- The checking of the termination criterion, implemented as a Spark reduce action (a distributed OR operation).

The two last steps are arranged into a loop that is executed until the termination criterion is met. After that the final selection of the best individual is also executed as a Spark reduce action (a distributed MIN operation).

The main issue found was the implementation of the mutation strategy because the population is partitioned and distributed among workers. For the mutation of each individual, random different individuals have to be selected from the whole population. How to access to individuals of other partitions from a given worker, having the constraint that only the driver has access to the complete population, was the main difficulty to be tackled. Three different variants have been considered for solving this problem:

- The driver distributes the random generation of keys to the workers, collects them, selects from the whole population the individuals corresponding to the generated random keys and distributes selected individuals to the workers that perform mutations and replacements.

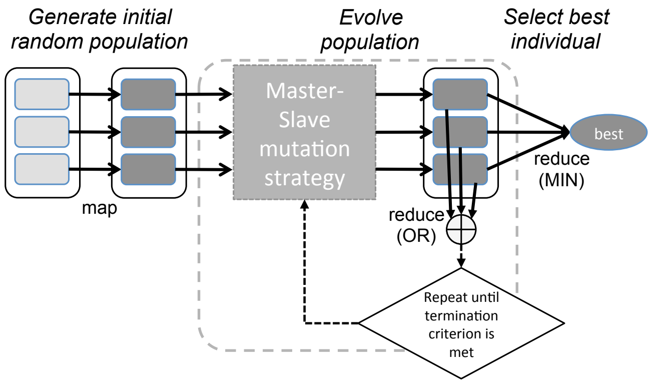

Generate initial random population
Evolve population
Select best individual
Master-Slave mutation strategy
map
best
reduce (MIN)
reduce (OR)
Repeat until termination criterion is met

Fig. 3: Spark-based Master-Slave implementation of the DE algorithm (`SmsPDE`).

- The driver itself makes the random pick of individuals for each member of the population and distribute them to the workers that perform mutations and replacements.
- The driver broadcasts the whole population to every worker using Spark *broadcast variables*. This Spark feature allows workers to have access to a local memory-cached read-only copy of the complete population. Therefore each worker can perform mutations picking the needed random individuals from its local copy of the population.

After benchmarking the performance of the three variants, broadcasting the population showed to be by far the best option. This is not surprising because the broadcasting feature of Spark is highly optimized and it is the recommended method for iterative algorithms to distribute data to workers that has to be reused by different iterations. Only the size of data to be broadcasted could discourage its use, but this is not the case with DE where the size of populations is small (usually in the range of 5D and 10D being D the problem dimension).

4.2 Island-based DE

When testing each one of the previous approaches, even using the version that has shown best benchmarking results, the penalty due to broadcast the whole population to workers in each iteration was unaffordable. For instance, using one of the benchmark functions used later on in Section 5, the f_{15} function, and a stopping criterion based on a predefined effort of 800,000 evaluations, the execution time of `seqDE` was 30 s, while the execution time of `SmsPDE` using 4 nodes was 263 s. The main conclusion of our experience with the master-slave implementation of the DE algorithm was that this approach does not fit well with the distributed model of Spark. Therefore we decided to implement a new parallel version of the algorithm using an island-based approach which in advance seemed to be a more promising one.

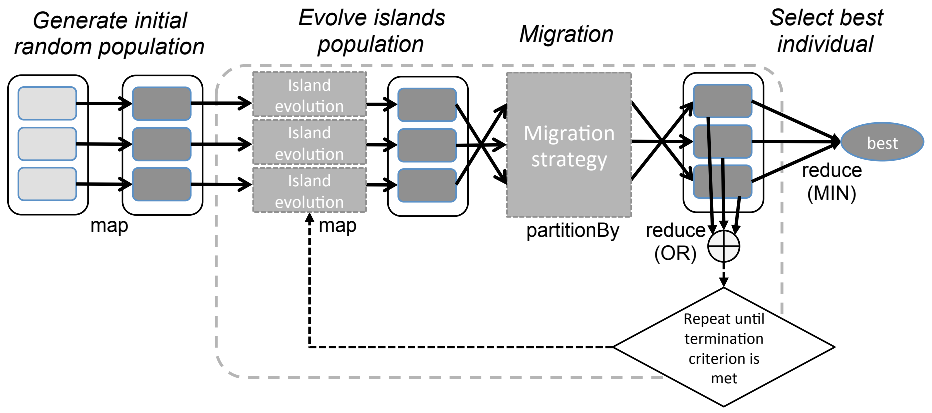


Fig. 4: Spark-based island implementation of the DE algorithm (`SiPDE`).

Figure 4 shows the scheme of the Spark-based island DE (`SiPDE`) implementation. As it can be seen it has some steps in common with the master-slave implementation: i.e. generation of the population, checking of the termination criterion and selection of the best individual. The main difference resides in the way the population evolves. Every partition of the population RDD has been considered to be an island, all with the same number of individuals. Islands evolve isolated during a number of evolutions. This number can be configured and is the same for all islands. During these evolutions every worker calculates mutations picking random individuals from its local partition only. To introduce diversity a migration strategy that exchanges selected individuals among islands is executed every time the number of evolutions is reached. This evolution-migration loop is repeated until the termination criterion is met.

For implementing the migration strategy a Spark feature known as *partitioner* has been used. In Spark the partitioner is responsible for assigning key-value pair RDD elements to partitions based on their keys. Default partitioner implements a hash-based partitioning using the Java hash code of the key. For this work we have implemented a custom partitioner that randomly and evenly shuffles elements among partitions. It must be noted that this partitioner leads to a migration strategy that randomly shuffles individuals among subpopulations without replacement. This partitioner proposal is intended to evaluate the migration communications overhead and not to improve the searching quality of the algorithm. Adding migration strategies with that purpose in mind are left for future work.

5 Experimental Results

In order to evaluate the efficiency of the Spark-based parallel implementation of the island DE algorithm (`SiPDE`), different experiments have been carried out. Its behavior, in terms of convergence and total execution time, has been compared with the sequential implementation (`seqDE`). For the experimental testbed Spark was deployed with default settings in the AWS public cloud using virtual clusters formed by 2, 4, 8 and 16 nodes communicated by the AWS standard network (Ethernet 1GB). For the nodes the `m3.medium` instance (1 vCPU, 3.75GB RAM, 4GB SSD) was used. Each experiment was executed a number of 10 independent runs on every virtual cluster, and the average and standard deviation of the execution time are reported in this section. It must be noted that, since Spark runs on the Java Virtual Machine (JVM), usual precautions (i.e. warm-up phase, effect of garbage collection) has been taken into account to avoid distortions on the measures.

The performed experiments used two sets of benchmark problems: a set of problems out of an algebraic black-box optimization testbed, the Black-Box Optimization Benchmarking (BBOB) data set [24]; and a challenging parameter estimation problem in systems biology [25]. On the one hand, the experiments over the BBOB data set were carried out to evaluate the efficiency of the proposed parallelization in a popular and accessible benchmarking testbed. On the other hand, the aim of the experiments with the parameter estimation in systems biology is to demonstrate the potential of the proposed techniques for improving the convergence and execution time of very hard problems. In these benchmarks, the execution of `seqDE` can take hours or even days to complete one only test. Four well known benchmarks problems from the BBOB data set were evaluated: Rastringin function (f_{15}), Schaffers function (f_{17}), Schwefel function (f_{20}), and Gallagher's Gaussian 21-hi Peaks function (f_{22}). The considered benchmark from the domain of computational system biology was a parameter estimation problem in a dynamic model of the *circadian* clock in the plant *Arabidopsis thaliana*, as presented in [25]. It must be noted that, as already available implementations in C/C++ and/or FORTRAN existed for all the benchmarks, we have wrapped them in our Scala code by using Java/Scala native interfaces (i.e JNI, JNA, SNA).

There are many configurable parameters in the classical DE algorithm, such as the mutation scaling factor (F), the crossover constant (CR) or the mutation strategy (MSt), whose selection may have a great impact in the algorithm performance. Since the objective of this work is not to evaluate the impact of these parameters, only results for one configuration are reported here. For the selection of the settings in these experiments, in general, the suggestions in [8] have been followed. Previous tests have been done to select those parameters that lead to reasonable computation times. Table 1 shows the selected configuration for each benchmark.

Comparing the sequential and the parallel metaheuristics is not an easy task, therefore, guidance of [24, 26] has been followed when analyzing the results of

Table 1: Benchmark functions. Parameters: dimension (D), population size (NP), crossover constant (CR), mutation factor (F), mutation strategy (MSt), value-to-reach/*ftarget* (VTR).

B	**Function**	**D**	**NP**	**CR**	**F**	**MSt**	**VTR**
BBOB benchmarks							
f_{15}	Rastrigin Function	5	800	.8	.9	DE/rand/1	1000
f_{17}	Schaffers F7 Function	6	1024	.8	.9	DE/rand/1	-16.94
f_{20}	Schwefel Function	6	1024	.8	.9	DE/rand/1	-546.5
f_{22}	Gallagher's Gaussian	10	1600	.8	.9	DE/rand/1	-1000
Systems Biology benchmark							
circadian	Circadian model	13	640	.8	.9	DE/rand/1	1e-5

these experiments. On the one hand, the behavior of the proposed solution was compared with the sequential classic version of DE (`seqDE`), therefore, speedups calculated as T_{seqDE}/T_{SiPDE} are reported in this section. On the other hand, both vertical and horizontal views can be used when evaluating a parallel metaheuristic. A vertical view assesses the performance of a fix number of evaluations, i.e., a pre-defined effort; while an horizontal view assesses the performance by measuring the time needed to reach a given target value. Thus, two different stopping criteria were considered in these experiments: maximum effort, for the vertical view, and solution quality (using as stopping criterion a Value-To-Reach), for the horizontal view.

Results from both views are shown in Table 2. For each experiment, the number of nodes ($\#n$) used, the mean execution time and the standard deviation (in seconds) of the 10 independent runs in each experiment, the average number of migrations ($\#m$) performed in the `SiPDE` method, and the speedup (sp) achieved are shown. It should be noted that the stopping criterion is evaluated during each island evolution but, when it is met by one or more islands, the algorithm only stops after the reduce operation at the end of the stage (see Figure 4). Thus, because no communication among workers is possible in Spark, the parallel `SiPDE` implementation cannot stop just right when the stopping criterion is reached (as the serial one does).

The figures for the vertical view (predefined effort) show that the proposed `SiPDE` method accelerates the computation of `seqDE` by performing the same number of evaluations in parallel. The figures for the horizontal view (quality solution) also demonstrate that the proposed `SiPDE` method reduces the computation time needed to achieve the VTR of the `seqDE` by improving the convergence of the algorithm.

The speedup achieved when using the predefined effort as stopping criterion deviates from the ideal one because of the overhead introduced by the communications. This fact can be observed in Figure 5 where both the speedup and efficiency are shown, the latter calculated as *speedup/np*. In the experiments with the BBOB benchmarks, due to their short execution times, only two migrations were performed among islands before the stopping criterion is met, while for

			Predefined Effort			Quality Solution		
	method	**#n**	**time±std**	**#m**	**sp**	**time±std**	**#m**	**sp**
f_{15}	`seqDE`	1	35.71 ± 0.20	-	-	47.89 ± 1.67	-	-
	`SiPDE`	2	27.86 ± 0.24	2	1.28	34.26 ± 1.47	2	1.40
		4	12.34 ± 0.21	2	2.89	15.88 ± 0.45	2	3.02
		8	6.39 ± 0.21	2	5.59	9.09 ± 0.25	2	5.27
		16	4.40 ± 0.40	2	8.12	3.73 ± 0.52	1.7	12.85
f_{17}	`seqDE`	1	36.60 ± 0.08	-	-	65.42 ± 2.13	-	-
	`SiPDE`	2	31.63 ± 0.27	2	1.16	53.74 ± 1.01	2	1.22
		4	13.31 ± 0.17	2	2.75	22.72 ± 0.28	2	2.88
		8	6.80 ± 0.24	2	5.38	12.79 ± 0.43	2	5.11
		16	4.25 ± 0.36	2	8.62	6.11 ± 0.35	2	10.71
f_{20}	`seqDE`	1	34.48 ± 0.07	-	-	55.12 ± 1.86	-	-
	`SiPDE`	2	27.00 ± 0.15	2	1.28	45.48 ± 2.19	2	1.21
		4	12.64 ± 0.16	2	2.73	17.78 ± 1.02	2	3.10
		8	6.47 ± 0.26	2	5.33	9.90 ± 1.05	2	5.57
		16	4.16 ± 0.41	2	8.28	4.08 ± 0.54	1.7	13.49
f_{22}	`seqDE`	1	112.14 ± 0.95	-	-	598.38 ± 478.31	-	-
	`SiPDE`	2	101.18 ± 0.94	2	1.11	703.97 ± 465.52	12.6	0.85
		4	44.03 ± 0.46	2	2.55	155.32 ± 156.19	7.8	3.85
		8	19.00 ± 0.40	2	5.90	84.60 ± 74.77	9.4	7.07
		16	10.25 ± 0.39	2	10.94	48.07 ± 37.64	11	12.45
circadian	`seqDE`	1	6267.97 ± 76.26	-	-	84482.53 ± 2369.75	-	-
	`SiPDE`	2	3111.14 ± 33.80	20	2.01	41736.17 ± 2114.45	26.7	2.03
		4	1575.26 ± 15.56	20	3.98	19029.74 ± 1042.31	24	4.44
		8	799.65 ± 9.23	20	7.84	8247.04 ± 558.07	20.4	10.24
		16	412.90 ± 4.93	20	15.18	2799.53 ± 377.43	13.4	30.39

Table 2: Execution time in seconds, number of migrations (*#m*), and speedup (*sp*) results for predefined effort (stopping criterion: $Nevals_{f15} = 1,000,000$; $Nevals_{f17} = 1,048,576$; $Nevals_{f20} = 1,040,000$; $Nevals_{f22} = 3,200,000$; $Nevals_{circadian} = 1,280,000$) and for a given solution quality (*Value-To-Reach* reported in table 1), using different number of nodes (*#n*). Average results from 10 independent runs in each experiment.

the *circadian* 20 migrations were performed. The efficiency results show that the overhead of the migrations barely affects on the performance when the execution time between two of them is significant (case of the *circadian* benchmark, where the efficiency is above 0.9), but it may greatly impact if it is small (case of BBOB benchmarks, where the efficiency is below 0.8 and significatively decreases when the number of nodes grows).

Figure 5 also shows the speedup and efficiency when using the quality of the solution as stopping criterion. Speedups are larger than the ones obtained for predefined effort because the cooperation among islands in the parallel searches modifies the systemic properties of the algorithm, improving its convergence and outperforming the serial one. In the figures reported in Table 2 it can be seen that

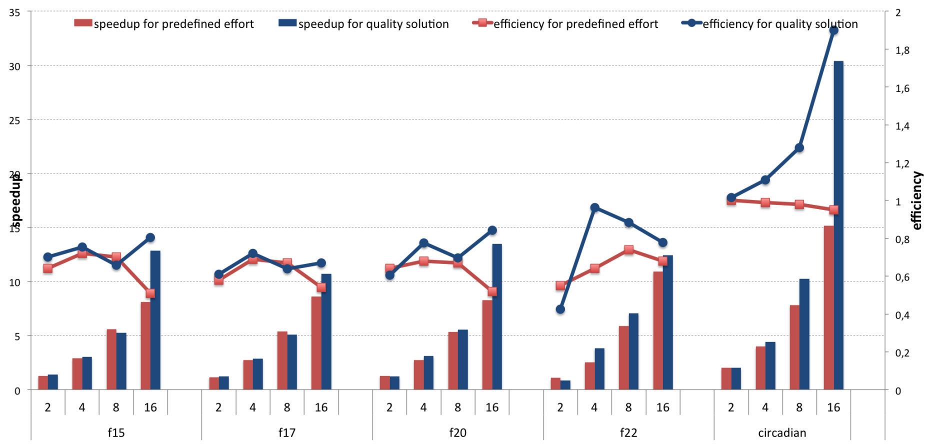


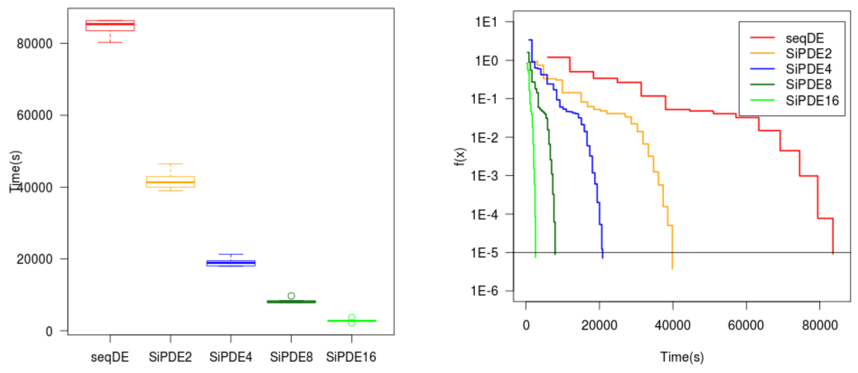
Fig. 5: Speedup and efficiency for results in Table 2.

for the f_{15}, f_{17} and f_{20} benchmarks, again due to their short execution times, most of them converged after two migrations. However, when the number of nodes grew, some benchmarks required only one migration (so giving an average of 1.7 in some experiments), thus, improving the efficiency obtained for 16 nodes. The f_{22} benchmark is a highly multimodal function that frequently fall into an undesired stagnation condition. Thus, the dispersion of the experimental results is very large (see the standard deviation in Table 2) and the number of migrations also varies from a minimum of 3 to a maximum of 19 in different runs.

For complex problems, like the *circadian* benchmark, the number of migrations clearly decreases with the number of nodes, demonstrating the potential of the parallel algorithm for improving the convergence of the DE method. The harder the problem is, the most improvement is achieved by the parallel algorithm, since the diversity introduced by the migration phase, although using a naive strategy as explained in Section 4.2, actually improves the effectiveness of the DE algorithm. Thus, for the *circadian* benchmark superlinear speedups are obtained, as well as efficiency above 1.

In this kind of stochastic problems it is also important to evaluate the dispersion of the experimental results. Figure 6(a) illustrates how the proposed `SiPDE` method reduces the variability of the DE execution time. This is an important feature that can be used to more accurately predict the boundaries in the cost of resources when using a public cloud like AWS.

Finally, to better illustrate the improvement of the proposed `SiPDE` method versus the `seqDE` method, Figure 6(b) shows, for the circadian benchmark, the convergence curves for different number of nodes. As expected, convergence time is considerably reduced by `SiPDE` with respect to `seqDE`. It must be noted that these results could be further improved using more skilled mutation and migra-



(a) Execution times (b) Convergence curves

Fig. 6: Box plot of the execution times and convergence curves for the *circadian* benchmark with quality solution stopping criterion.

tion strategies and adding enhancements, like local search [27], which have not been considered in this work.

6 Conclusions and Future Work

In order to explore how parallel metaheuristics could take advantage of the recent advances in Cloud programming models, in this paper Spark-based implementations of two different parallel schemes of the Differential Evolution (DE) algorithm, the master-slave and the island-based, are proposed and evaluated. Early benchmarking results showed that the island-based solution is by far the best suited to the distributed nature of Spark. Thus a thorough evaluation of this implementation was conducted on the AWS public cloud using a real testbed consisting on virtual clusters of different sizes.

Both synthetic and real biology-inspired benchmarks were used for the evaluation. The experimental results show that the proposal achieves not only a competitive speedup against the serial implementation, but also a good scalability when the number of nodes grows. The paper can be useful for those interested in the potential of Spark in computationally intensive nature-inspired methods in general and DE in particular. To the best of our knowledge, this is the first work on using Spark to parallelize DE.

Future work will be focus on developing new migration strategies and including further optimizations to improve the convergence of the Spark-based island parallel DE proposed.

Acknowledgements

This research received financial support from the Spanish Ministerio de Economía y Competitividad (and the FEDER) through the Project SYNBIOFACTORY (grant number DPI2014-55276-C5-2-R). It has been also supported by the Spanish Ministerio de Ciencia e Innovación (and the FEDER) through the Project TIN2013-42148-P, and by the Galician Government (Xunta de Galicia) under the Consolidation Program of Competitive Research Units (Network Ref. R2014/041 and Project Ref. GRC2013/055) cofunded by FEDER funds of the EU.

References

1. Floudas, C.A., Pardalos, P.M.: Optimization in computational chemistry and molecular biology: local and global approaches. Volume 40. Springer Science & Business Media (2013)
2. Banga, J.R.: Optimization in computational systems biology. BMC systems biology **2**(1) (2008) 47
3. Grossmann, I.E.: Global optimization in engineering design. Volume 9. Springer Science & Business Media (2013)
4. Crainic, T.G., Toulouse, M.: Parallel strategies for meta-heuristics. Springer (2003)
5. Alba, E.: Parallel metaheuristics: a new class of algorithms. Volume 47. Wiley-Interscience (2005)
6. Dean, J., Ghemawat, S.: MapReduce: simplified data processing on large clusters. In: Proceedings of the 6th USENIX Symposium on Operating Systems Design and Implementation, OSDI'04. (2004)
7. Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauly, M., Franklin, M.J., Shenker, S., Stoica, I.: Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing. In: Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2012. (2012)
8. Storn, R., Price, K.: Differential evolution - a simple and efficient heuristic for global optimization over continuous spaces. Journal of Global Optimization **11**(4) (1997) 341–359
9. Alba, E., Luque, G., Nesmachnow, S.: Parallel metaheuristics: recent advances and new trends. International Transactions in Operational Research **20**(1) (2013) 1–48
10. McNabb, A.W., Monson, C.K., Seppi, K.D.: Parallel PSO using MapReduce. In: IEEE Congress on Evolutionary Computation, CEC2007, IEEE (2007) 7–14
11. Jin, C., Vecchiola, C., Buyya, R.: MRPGA: an extension of MapReduce for parallelizing genetic algorithms. In: IEEE Fourth International Conference on eScience, eScience'08, IEEE (2008) 214–221
12. Verma, A., Llora, X., Goldberg, D.E., Campbell, R.H.: Scaling genetic algorithms using MapReduce. In: Ninth International Conference on Intelligent Systems Design and Applications, ISDA'09, IEEE (2009) 13–18
13. Radenski, A.: Distributed simulated annealing with MapReduce. In: Applications of Evolutionary Computation. Springer (2012) 466–476
14. Lee, W.P., Hsiao, Y.T., Hwang, W.C.: Designing a parallel evolutionary algorithm for inferring gene networks on the cloud computing environment. BMC systems biology **8**(1) (2014) 5

15. Zhou, C.: Fast parallelization of differential evolution algorithm using MapReduce. In: Proceedings of the 12th Annual Conference on Genetic and Evolutionary Computation, ACM (2010) 1113–1114
16. Tagawa, K., Ishimizu, T.: Concurrent differential evolution based on MapReduce. International Journal of Computers **4**(4) (2010) 161–168
17. Daoudi, M., Hamena, S., Benmounah, Z., Batouche, M.: Parallel differential evolution clustering algorithm based on MapReduce. In: 6th International Conference of Soft Computing and Pattern Recognition (SoCPaR), IEEE (2014) 337–341
18. Ekanayake, J., Li, H., Zhang, B., Gunarathne, T., hee Bae, S., Qiu, J., Fox, G.: Twister: a runtime for iterative MapReduce. In: The First International Workshop on MapReduce and its Applications. (2010)
19. Zhang, Y., Gao, Q., Gao, L., Wang, C.: IMapReduce: a distributed computing framework for iterative computation. In: Proceedings of the 1st International Workshop on Data Intensive Computing in the Clouds (DataCloud. (2011) 1112
20. Bu, Y., Howe, B., Balazinska, M., Ernst, M.D.: HaLoop: efficient iterative data processing on large clusters
21. Ewen, S., Tzoumas, K., Kaufmann, M., Markl, V.: Spinning fast iterative data flows. CoRR **abs/1208.0088** (2012)
22. Alexandrov, A., Bergmann, R., Ewen, S., Freytag, J.C., Hueske, F., Heise, A., Kao, O., Leich, M., Leser, U., Markl, V., Naumann, F., Peters, M., Rheinländer, A., Sax, M., Schelter, S., Höger, M., Tzoumas, K., Warneke, D.: The stratosphere platform for big data analytics. The VLDB Journal **23**(6) (2014) 939–964
23. Odersky, M., Micheloud, S., Mihaylov, N., Schinz, M., Stenman, E., Zenger, M., et al.: An overview of the Scala programming language. Technical report (2004)
24. Hansen, N., Auger, A., Finck, S., Ros, R.: Real-parameter black-box optimization benchmarking 2009: experimental setup. Technical Report RR-6828, INRIA (2009)
25. Locke, J., Millar, A., Turner, M.: Modelling genetic networks with noisy and varied experimental data: the circadian clock in arabidopsis thaliana. Journal of Theoretical Biology **234**(3) (2005) 383–393
26. Alba, E., Luque, G.: Evaluation of parallel metaheuristics. In: PPSN-EMAA'06, Reykjavik, Iceland (September 2006) 9–14
27. Penas, D., Banga, J., González, P., Doallo, R.: Enhanced parallel differential evolution algorithm for problems in computational systems biology. Applied Soft Computing **33** (2015) 86 – 99