Evaluation of parallel Differential Evolution implementations on MapReduce and Spark

Diego Teijeiro[1], Xoán C. Pardo[1], David R. Penas[2], 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. Global optimization problems arise in many areas of science and engineering, computational and systems biology and bioinformatics among them. Many research efforts have focused on developing parallel metaheuristics to solve them in reasonable computation times. Recently, new programming models are being proposed to deal with large scale computations on commodity clusters and Cloud resources. In this paper we investigate how parallel metaheuristics deal with these new models by the parallelization of the popular Differential Evolution algorithm using MapReduce and Spark. The performance evaluation has been carried out both in a local cluster and in the Amazon Web Services public cloud. The results obtained can be particularly useful for those interested in the potential of new Cloud programming models for parallel metaheuristic methods in general and Differential Evolution in particular.
Keywords: Parallel Metaheuristics, Differential Evolution, Cloud Computing, MapReduce, Spark

1 Introduction

Many key problems in computational systems biology can be formulated and solved using global optimization techniques. Metaheuristics are gaining increased attention as an efficient way of solving hard global optimization problems. Differential Evolution (DE) [1] is one of the most popular heuristics for global optimization, and it has been successfully used in many different areas [2]. However, in most realistic applications, like parameter estimation problems in systems biology, this population-based method requires a very large number of evaluations (and therefore, large computation time) to obtain an acceptable result. Therefore, several parallel DE schemes have been proposed, most of them focused on traditional parallel programming interfaces and infrastructures.

The aim of this paper is to investigate how parallel metaheuristics could be handled based on the recent advances in Cloud programming models. Distributed frameworks like MapReduce or Spark, provide advantages such as higher-level programming models to easily parallelize user programs, support for data distribution and processing on multiple nodes/cores, and run-time features such as

fault tolerance and load-balancing. The goal of this paper is to explore this direction further considering a parallel implementation of DE in both frameworks and evaluating their performance in a real testbed using both a local cluster and the Amazon Web Services (AWS) public cloud.

2 Background and Related Work

Since its appearance, MapReduce [3] (MR from now on) has been the distributed programming model for processing large scale computations that has attracted more attention. In short, MR 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 MR are made in batches, using a distributed filesystem to take the input and store the output. MR has been applied to a wide range of applications, including distributed sorting, graph processing or machine learning. But for iterative algorithms, as those typical in parallel metaheuristics, MR has shown serious performance bottlenecks [4] because there is no way of reusing data or computation from previous iterations efficiently when several of these single batches are executed inside a loop.

Spark [5] is a recent proposal designed from the very beginning to provide efficient support for iterative algorithms. Spark provides a distributed memory abstraction denominated *resilient distributed datasets* (RDDs) for supporting fault-tolerant and efficient in-memory computations. Formally, an RDD is a read–only fault–tolerant partitioned collection of records. RDDs are created from other RDDs or from data in stable storage by applying coarse-grained *transformations* (e.g., *map*, *filter* or *join*) that can be pipelined to form a *lineage*. 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. Spark runtime is composed of a single *driver* program and multiple long-lived *workers* that persist RDD partitions in RAM across operations. Developers write the driver program where they define one or more RDDs and invoke actions on them. Lineages are used to compute RDDs lazily whenever they are used in actions or to recompute them in case of failure.

There are some proposals which investigate how to apply MR to parallelize the DE algorithm. In [6] the Hadoop framework (the most widely used open source implementation of MR) is used to perform in parallel the fitness evaluation. However, the experimental results reveal that HDFS (Hadoop Distributed File System) I/O and system bookkeeping overhead significantly reduces the benefits of the parallelization. In [7], a concurrent implementation of the DE based on MR is proposed which was only evaluated on a multi-core CPU taking advantage of the shared-memory architecture. In [8] a parallel implementation of DE based clustering using MR is also proposed. This algorithm was implemented in three levels, each of which consists of DE operations. The use of Spark for the parallelization of the DE algorithm was explored in [12]. In that paper Spark-based implementations of two different parallel schemes of the DE algorithm, the master-slave and the island-based, are proposed and evaluated. Results showed

that the island-based solution is by far the best suited to the distributed nature of Spark.

There are also a few references comparing MR and Spark performance for iterative algorithms. In [5] Spark authors compare their proposal to MR (using Hadoop) for different types of iterative algorithms on Amazon EC2. They implemented two machine learning algorithms, logistic regression which is more I/O-intensive and k-means which is more compute-intensive, and found that scaling up to 100 nodes Spark outperformed MR from 12.3x up to 25.3x for logistic regression and from 1.9x up to 3.2x for k-means. They also tested the well-known PageRank algorithm finding that Spark outperformed MR by up to 7.4x, scaling well in up to 60 nodes. In [9] performance of several distributed frameworks including Hadoop (v.1.0.3) and Spark (v.0.8.0) were assessed in Amazon EC2 for iterative scientific algorithms. The Partitioning Around Medoids (PAM) clustering algorithm and the Conjugate Gradient (CG) linear system solver were implemented for evaluation. Results show that scaling up to 32 nodes and using 3 datasets of different sizes, Spark outperformed MR from 1.3x up to 48x for PAM and from 23x up to 99x for CG. Authors concluded that Spark results seemed to be greatly affected by the characteristics of the benchmarking algorithms and their dataset composition. In [10] Hadoop (v.2.4.0) and Spark (v.1.3.0) major architectural components are thoroughly compared using a set of analytic workloads. Results show that, using a 4 node (32 cores, 190 GB RAM, 9x1TB disk each) cluster with a 1GB Ethernet network, Spark outperformed MR by 5x for k-means, linear regression and PageRank. Authors conclude that, for iterative algorithms, caching the input as RDDs in Spark can reduce both CPU and disk I/O overheads for subsequent iterations and that RDD caching is much more efficient than other low-level caching approaches such as OS buffer caches, and HDFS caching, which can only reduce disk I/O.

3 Implementing DE on MR and Spark

Differential Evolution [1] 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 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.

However, typical runtimes for many realistic problems are in the range from hours to days due to the large number of objective function evaluations needed, making the performance of the classical sequential DE unacceptable. In the literature, different parallel models can be found [11] aiming to improve both computational time and number of iterations for convergence. The *master-slave*

Algorithm 1: Differential Evolution algorithm

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
        a⃗, b⃗, c⃗ ⇐ different random individuals from P matrix
        for k ⇐ 0 to D do
            if random point is less than CR then
                Ind⃗(k) ⇐ a⃗(k) + F(b⃗(k) - c⃗(k))
            end
        end
        if Evaluation(Ind⃗) is better than Evaluation(P(x)⃗) then
            Replace_Individual(P,Ind⃗)
        end
    end
until Stop conditions;
```

and the *island-based* models are the most popular. In the *master-slave* model the behavior of the sequential DE is preserved by parallelizing the inner-loop of the algorithm, where a master processor distributes computation between the slave processors. 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.

The implementation of the DE master-slave model does not fit well with the distributed nature of programming models like MR or Spark [12]. The reason is that when the mutation strategy is applied to each individual, random different individuals have to be selected from the whole population. Considering that the population would certainly be partitioned and distributed among slaves, any solution to this problem would introduce an unfeasible communications overhead. In the rest of this section we briefly describe our island-based parallel implementations of the DE algorithm, which in advance seemed to be a more promising approach, for both MR and Spark.

Algorithm 2 shows the pseudocode for the *driver* (the user-provided code run by the master) of our island-based parallel implementation of the DE algorithm using MR. The driver is responsible for randomly generating the initial population and for evolving it repeating a loop until the termination criterion is met. In each loop iteration the population is randomly partitioned into islands all with the same number of individuals, islands are written to HDFS one file each, a MR job for evolving the islands is configured and launched and the evolved global population is gathered from HDFS after the MR job finished. Algorithm 3 shows the pseudocode of the map functions executed in each MR job. Each map is responsible for the evolution of exactly one island isolated from the rest dur-

Algorithm 2: Driver pseudocode

input : DE configuration parameters
output: A population P whose individuals were optimized

$P \Leftarrow$ initial random population
$\#i \Leftarrow$ number of islands

repeat
  $\overrightarrow{Islands} \Leftarrow$ PartitionPopulation(P, $\#i$) `// with shuffling`
  $P \Leftarrow$ EvolveIslands($\overrightarrow{Islands}$) `//the MR job`
until *Stop conditions*;

Algorithm 3: Map pseudocode

inputs : An island I; DE configuration parameters
output: An island I whose individuals were optimized

repeat
  I $\Leftarrow$ EvolveIsland(I) `// apply the DE mutation strategy`
until *number of evolutions*;

for *each individual* $\overrightarrow{Ind}$ *of the island* I **do**
  Emit(Evaluation($\overrightarrow{Ind}$), $\overrightarrow{Ind}$)
end

ing a predefined number of evolutions, the same for all islands. The map starts by reading the island individuals from HDFS and storing them in local memory, then applying the DE mutation strategy taking random individuals only from its island until the predefined number of evolutions is reached and, finally, emitting an output record for each individual of the evolved island using its fitness value as key. The MR job implementation is completed with a single identity reducer which simply receives the individuals from all the islands ordered by their fitness value and writes them to an HDFS file. Note that, as individuals are ordered by fitness, the first record in the output file will be the best individual. To introduce diversity a migration strategy that randomly shuffles individuals among islands without replacement is applied by the driver during the partition of the population in islands. This is a naive strategy intended only to evaluate the migration overhead and not to improve the searching quality of the algorithm.

Figure 1 shows the scheme of our island-based parallel DE implementation using Spark. In the figure, boxes with solid outlines are RDDs. Partitions are shaded rectangles, darker if they are persisted in memory. A key-value pair RDD has been used to represent the population where each individual is uniquely identified by its key. The algorithm starts by distributing the random generation and initial evaluation of individuals that form the population using an Spark map transformation, then an evolution-migration loop is repeated until the termination criterion implemented as an Spark reduce action (a distributed

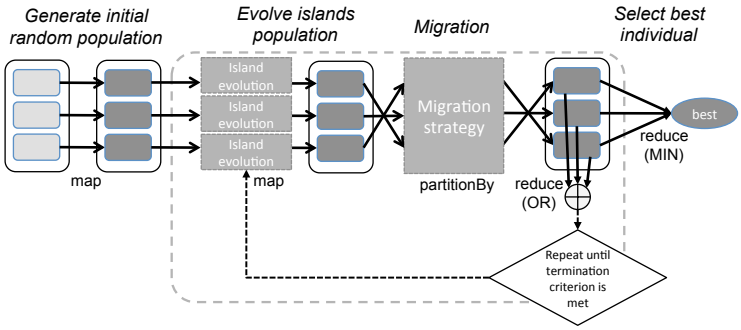

Fig. 1: Island-based parallel DE implementation using Spark.

OR operation) is met and, finally, the selection of the best individual is done by using an Spark reduce action (a distributed MIN operation). In the evolution-migration loop 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 predefined number of evolutions, the same for all islands, and in order to introduce diversity the same migration strategy as in the MR implementation is executed after an evolution. We have developed a custom Spark *partitioner* that randomly and evenly shuffles elements among partitions for implementing the migration strategy.

It must be noted that although the migration strategy is the same for both implementations, the overhead they add is not. In MR migration is implemented in the driver that reads the population from HDFS, shuffles the individuals among islands and writes back the islands to HDFS, so the overhead is mainly caused by accessing HDFS. In Spark migration is implemented as a *partitionBy* operation, so the overhead is mainly caused by communications.

4 Experimental Results

In order to evaluate and compare the island-based implementation of DE using MR and Spark, different experiments have been carried out. Their behavior, in terms of execution time and overhead, has been compared with the sequential implementation. Programming languages used have been Scala (v2.10) for the sequential and Spark implementations, and Java (v1.7.0) for the MR implementation. Spark (v.1.4.1) and Hadoop (v2.7.1) frameworks were used for the experiments.

Two sets of benchmark problems were used: on the one hand, two problems out of an algebraic black-box optimization testbed, the Black-Box Optimization Benchmarking (BBOB) data set [13]: Rastringin function (f_{15}) and Gallagher's Gaussian 21-hi Peaks function (f_{22}); on the other hand, a challenging parameter

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
f_{15}	Rastrigin Function	5	1024	.8	.9	DE/rand/1	1000
f_{22}	Gallagher's Gaussian	10	1600	.8	.9	DE/rand/1	-1000
circadian	Circadian model	13	640	.8	.9	DE/rand/1	1e-5

estimation problem in a dynamic model of the *circadian* clock in the plant *Arabidopsis thaliana*, as presented in [14]. Table 1 shows the configurable parameters used for the reported experiments.

For the experimental testbed two different platforms has been used. First, experiments were conducted in our multicore local cluster Pluton, that consists of 16 nodes powered by two octa-core Intel Xeon E5-2660 CPUs with 64 GB of RAM, and connected through an InfiniBand FDR network. Second, experiments were 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. In both testbeds, each experiment was executed a number of 10 independent runs, and the average and standard deviation of the execution time are reported in this section. Note that, since Spark and MR programs run on the Java Virtual Machine (JVM), usual precautions (i.e. warm-up phase, effect of garbage collection) have been taken into account to avoid distortions on the measures.

Comparing the sequential and the parallel metaheuristics is not an easy task, therefore, guidance of [13, 15] has been followed when analyzing the results of these experiments. Since the parallel strategy followed is the same in both MR and Spark implementations, the best way to fairly compare the performance of both implementations is to stop at a predefined effort, that is, for a vertical view. Results obtained in the local cluster Pluton and in the AWS public cloud, both in terms of execution times and speedups, are shown in Figure 2 using a predefined number of evaluations as stopping criterion. All the experiments execute two iterations of the algorithm (each iteration corresponding to an evolution-migration). To assess the scalability up to 16 islands have been used for the parallel implementations. We do not use more than 16 islands due to the small population size in these benchmarks. As it can be seen, the Spark implementation achieves good results, both in time and speedup, versus the sequential algorithm, and a good scalability when the number of islands grows. However, the MR implementation presents poorer results, specially for f_{15}, the shortest of the two benchmarks, because it introduces a high overhead. Note that the execution times are larger in AWS than in cluster Pluton, even for sequential executions. Virtualization overhead, use of non-dedicated resources in a multi-tenant platform, and differences in node characteristics can explain these results. Even so, the Spark implementation achieves good results in terms of speedup

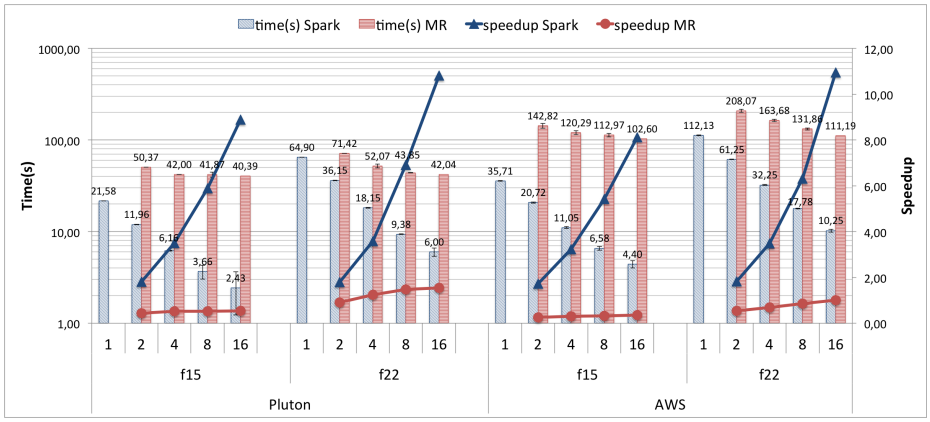

Fig. 2: Execution time and speedup results comparing MR vs Spark implementation in: (a) cluster Pluton (b) the AWS public cloud. Stopping criterion: $Nevals_{f15} = 1,025,024$ and $Nevals_{f22} = 3,200,000$.

versus the sequential implementation in AWS. However, the MR implementation presents even poorer results than in the experiments carried out in the local cluster.

To evaluate the overhead introduced by MR and Spark we have used modified versions of our implementations in which the evolution of the population was removed. Each modified implementation was executed for a total of 8 *evolution-migration* iterations and the overhead of each iteration was measured separately in order to assess differences between them. Figure 3 shows the results obtained. The first iteration in the Spark implementation is always the most time consuming (it corresponds to the outliers in the box plots). However, the rest of the iterations present even lower overhead and lower dispersion in the results, being the mean overhead of each iteration of 0.023s. In the case of MR there is no significant difference between iterations, and the figures clearly indicate a higher overhead and large dispersion in the results, being the mean overhead of each iteration 17.95s in Pluton. This explains why, in Figure 2, execution times of MR implementation stagnate around 40s (close to the overhead of the two iterations) when the number of cores grows.

Figure 3 shows also that, both for MR and Spark, the overhead in AWS is larger than in the local cluster. The first iteration in Spark is again the most time consuming. The rest of the iterations present low overheads, being the mean overhead of each iteration of 0.09s. Also, it must be noted that the Spark overhead slightly increases when the number of nodes grows. Results in the local cluster does not clearly show this increase, but it should be noted that differences are very small and we are shuffling very few data among a small number of physically close nodes using a high-throughput and low-latency InfiniBand network. In the case of MR in AWS there is again no significant difference between the first and the subsequent iterations, but overheads are higher than in

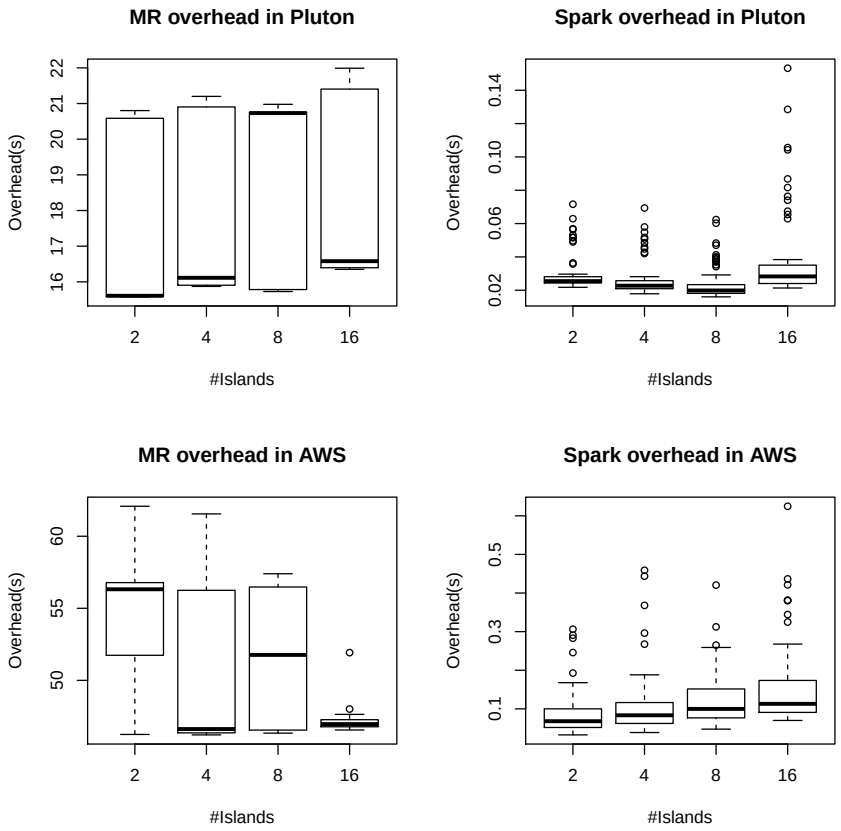

Fig. 3: Box plot of the overhead times per evolution-migration iteration in MR and Spark.

the local cluster. In addition to the overhead due to the virtualization and the differences in node characteristics, these results could be explained by the use of HDFS with Amazon EBS volumes which are mounted over a non-dedicated 1GB Ethernet network. These boxplots also show that the variability in the MR overhead between independent experiments is much more noticeable in AWS. For instance, experiments with 2, 4, 8 and 16 nodes were performed at different moments and, although they obtain similar mean overhead, the standard deviation is significantly different.

Previous results explain why Spark outperforms MR for short execution benchmarks. In order to honestly evaluate the performance in long real applications, we have considered a parameter estimation problem, the *circadian* benchmark, and we have used as stopping criterion a *value-to-reach* to assess the performance form an horizontal view. Figure 4 shows a bean plot that al-

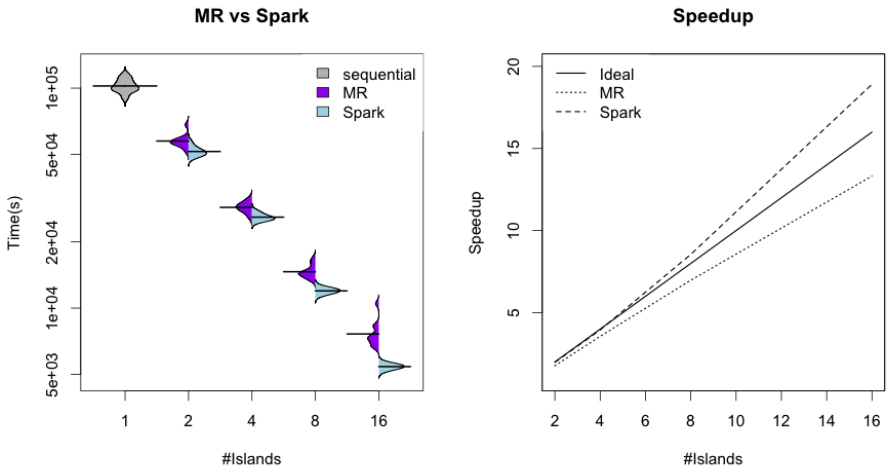

Fig. 4: Circadian benchmark. Comparing MR vs Spark implementation in cluster Pluton: (a) bean plots of the execution time, (b) speedup results vs the sequential implementation.

lows for an easy comparison of the execution times obtained using the MR and the Spark implementations in the local cluster. Note that not only the execution time is larger for the MR implementation but also the dispersion of the results obtained is bigger. Figure 4 also shows the speedup achieved. The improvement of the parallel versions against the sequential one is due both to the distribution of computations among the workers and to the effectiveness of the parallel algorithm, since the diversity introduced by the migration phase actually reduces the number of evaluations required for convergence. The harder the problem is, the more improvement is achieved by the parallel algorithm. Thus, for the *circadian* benchmark, when using Spark, superlinear speedups are obtained. The MR implementation also achieves a reduction in the number of evaluations required when the number of islands grows, however, the overhead introduced by MR restrains it from attaining such speedups.

It must be noted that for problems with long execution times where the iteration selectivity (as defined in [10]) is very low, like it is the case for the DE algorithm, MR is favoured because the overhead accessing HDFS is very small. For long-execution applications, such as the circadian benchmark, where the computation time dominates the overhead introduced by the iterations in MR, MR is competitive with Spark, though the latter still presents better scalability (as shown in Figure 4), since increasing the number of resources decreases the computation time but, as we have seen, the overhead does not decrease.

Finally, although it is not the aim of this work, we have performed several preliminary tests to assess how competitive the Spark parallel implementation can be with respect to traditional HPC solutions. The same previous experi-

ments were carried out with the implementation of the asynchronous parallel DE described in [16]. This implementation is coded in C and uses the OpenMPI library. Directly comparing the execution times of both implementations is not fair, since the implemented algorithms are not the same: (i) the MPI implementation includes some heuristics to improve the convergence rate of the DE, and (ii) the migration strategy is not the same in both algorithms. Thus, we have estimated the execution time per evaluation such as $T_{eval} = T_{total}/N_{evals}$. Note that this estimated T_{eval} includes not only the CPU time for the evaluation itself but also the communication time and other overheads introduced by the algorithm implementation. We encountered that execution time per evaluation of the Spark implementation was between 2.24x and 2.57x the execution time per evaluation of the MPI implementation. 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 code by using Java/Scala native interfaces (i.e JNI, JNA, SNA). Further studies to determine a more accurate interpretation of this overhead are left for future work.

5 Conclusions

In order to explore how parallel metaheuristics could take advantage of the recent advances in Cloud programming models, in this paper MR and Spark island-based implementations of the DE algorithm are proposed and evaluated. The performance evaluation of both implementations was conducted on a local cluster and on the AWS public cloud. Both synthetic and real biology-inspired benchmarks were used for the testbed. Although this paper is focussed on the DE algorithm, we believe that both the description of the implementations and the results obtained in this work can be useful for those interested in the potential of cloud programming models for developing other parallel metaheuristic methods.

The experimental results show that MR has significant higher overhead per iteration than Spark mainly caused by longer task initialization times and HDFS access, and that Spark has best support for iterative algorithms as it reduces the overhead between the first and subsequent iterations. For short benchmarks Spark clearly outperforms MR, which speedup is limited by its overhead. For long running benchmarks, in which computation time prevails over iteration overhead, MR is competitive with Spark. In addition, MR would be favoured by algorithms with low iteration selectivity (i.e. small population size) like DE, but on the contrary, it would be harmed by algorithms with short iterations and higher iteration selectivity.

Acknowledgements

Financial support from the Spanish Government (and the FEDER) through the projects DPI2014-55276-C5-2-R, TIN2013-42148-P, and from the Galician Government 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. 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
2. Das, S., Suganthan, P.N.: Differential evolution: A survey of the state-of-the-art. Evolutionary Computation, IEEE Transactions on **15**(1) (2011) 4–31
3. Dean, J., Ghemawat, S.: MapReduce: simplified data processing on large clusters. In: The 6th USENIX Symposium on Operating Systems Design and Implementation. (2004)
4. 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)
5. Zaharia, M., et al.: Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing. In: The 9th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2012. (2012)
6. 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
7. Tagawa, K., Ishimizu, T.: Concurrent differential evolution based on MapReduce. International Journal of Computers **4**(4) (2010) 161–168
8. 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
9. Jakovits, P., Srirama, S.N.: Evaluating mapreduce frameworks for iterative scientific computing applications. In: International Conference on High Performance Computing & Simulation, HPCS 2014, IEEE (2014)
10. Shi, J., Qiu, Y., Minhas, U.F., Jiao, L., Wang, C., Reinwald, B., Özcan, F.: Clash of the titans: Mapreduce vs. spark for large scale data analytics. In: Proceedings of the Very Large Data Bases (VLDB) Endowment. Volume 8. (2015) 2110–2121
11. Alba, E., Luque, G., Nesmachnow, S.: Parallel metaheuristics: recent advances and new trends. International Transactions in Operational Research **20**(1) (2013) 1–48
12. Teijeiro, D., Pardo, X.C., González, P., Banga, J.R., Doallo, R.: Implementing Parallel Differential Evolution on Spark. In: Applications of Evolutionary Computation: 19th European Conference, EvoApplications 2016, Proceedings, Part II. (2016) 75–90
13. Hansen, N., Auger, A., Finck, S., Ros, R.: Real-parameter black-box optimization benchmarking 2009: experimental setup. Technical Report RR-6828, INRIA (2009)
14. 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
15. Alba, E., Luque, G.: Evaluation of parallel metaheuristics. In: PPSN-EMAA'06, Reykjavik, Iceland (September 2006) 9–14
16. Penas, D.R., Banga, J.R., González, P., Doallo, R.: Enhanced parallel differential evolution algorithm for problems in computational systems biology. Applied Soft Computing **33** (2015) 86 – 99