Implementing cloud-based parallel metaheuristics: an overview

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

[1] *Computer Architecture Group, Universidade da Coruña, Spain*
{patricia.gonzalez, xoan.pardo, doallo}@udc.es
[2] *BioProcess Engineering Group, IIM-CSIC, Spain*
julio@iim.csic.es

Abstract

Metaheuristics are among the most popular methods for solving hard global optimization problems in many areas of science and engineering. Their parallel implementation applying HPC techniques is a common approach for efficiently using available resources to reduce the time needed to get a good enough solution to hard-to-solve problems. Paradigms like MPI or OMP are the usual choice when executing them in clusters or supercomputers. Moreover, the pervasive presence of cloud computing and the emergence of programming models like MapReduce or Spark have given rise to an increasing interest in porting HPC workloads to the cloud, as is the case with parallel metaheuristics. In this paper we give an overview of our experience with different alternatives for porting parallel metaheuristics to the cloud, providing some useful insights to the interested reader that we have acquired through extensive experimentation.

Keywords: parallel metaheuristics, cloud computing, MPI, MapReduce, Spark

1 Introduction

Optimization is concerned with finding the "best available" solution for a given problem. Many key problems in different areas of science, economics and engineering can be formulated and solved using different optimization techniques [1, 2, 3]. For example, optimization problems are playing an increasing role in computational biology, bioinformatics and chemistry, helping in the development of novel therapies and drugs for different diseases such as cancer or autoimmune diseases. Most of these optimization problems are, in practice, NP-hard, complex, and time-consuming. Stochastic global optimization methods are robust alternatives to solve these problems. And among them, metaheuristics are gaining increased attention as an efficient way of solving hard global optimization problems. However, in most realistic applications, metaheuristics still require large computation time to obtain an acceptable result.

Thus, the parallelization of optimization methods in general, and of metaheuristics in particular, and the use of HPC resources, like clusters or supercomputers, have been a common approach to speed-up the computations, increase the size of the problems that can be handled or attempt a more thorough exploration of the solution space. Furthermore, the technological developments of the last decades, continuously reducing the cost/performance ratio of HPC resources, have made accessing to them more feasible. However, when considering very challenging problems, the provision of a large number of resources, which is not always practicable, is essential for the success of the parallel solution. The emergence in the last years of cloud computing [4] as a new model for the effortless provision of a large number of computing resources has attracted the attention of the HPC community.

Cloud computing has some specific characteristics that make it an interesting alternative for the provision and management of computing resources. Public cloud providers offer self-service interfaces to external users for the on-demand provision of virtualized computing resources. Users share the provider infrastructure while having exclusive zero-queue-time access to their own isolated virtual resources, configured to their applications' needs. Resource consumption is charged using a pay-as-you-go model at very appealing prices. Additionally, collaborators can be provided with controlled access to resources and environments be shared with third-parties using virtual machine images.

But despite these appealing features, its adoption by the HPC community has been limited. The main reason is the performance obtained when using traditional parallel programming models, like MPI, on virtual clusters. Several evaluations [5, 6] have shown that although computationally-intensive applications present little overhead, communication-intensive ones

have poor performance thus limiting their scalability.

On the other hand, to facilitate the development of large-scale distributed applications, new programming models like MapReduce [7] or Spark [8] have been proposed. These models combine high-level programming abstractions and distributed execution frameworks with implicit support for deployment, data distribution, parallel processing and run-time features like fault tolerance or load balancing. But, although their many advantages, these proposals also have some shortcomings that could discourage their use for HPC workloads. They are designed with the analytics of big amounts of data in mind, preferring availability to performance and providing lower speedup and distributed efficiency than traditional parallel frameworks.

In this paper we give an overview of our experience with the different alternatives we have studied for porting some representative parallel metaheuristics to the cloud. By means of extensive experimentation, using both synthetic and real-world benchmarks on different traditional and cloud testbeds, we have analyzed the pros and cons of the different options. Considering the demanding and dynamic nature of the implementation of these methods, we think that they are exemplary of the many applications that could benefit from being ported to the cloud, and that the conclusions drawn could be useful in many other scenarios.

The rest of the paper is structured as follows. Section 2 describes the alternatives for writing parallel programs for the cloud that we have analyzed. Section 3 introduces the Differential Evolution metaheuristic and the parallel versions we have implemented. In section 4 the main lessons learned from experimental results are overviewed and justified. Section 5 references some related work. Finally, Section 6 summarizes the conclusions of the paper.

2 Parallel programming in the cloud

Nowadays there are different alternatives that could be used to implement parallel programs for the cloud. Examples include "traditional" message-passing or many-task computing approaches, using actors or using data-oriented models and abstractions like MapReduce (MR) or Resilient Distributed Datasets (RDD). From these alternatives, we have evaluated and compared three different approaches. Each of them is based on different programming abstractions, implemented using different programming languages and executed using different platforms:

1. A C implementation using MPI in a virtual cluster.
2. A Java implementation using MR in a Hadoop cluster.
3. A Scala implementation using the RDD abstraction in a Spark cluster.

The **Message Passing Interface (MPI)** is the de-facto standard for HPC distributed computing. It is a standard interface that allows developers to write parallel applications in C, C++ or FORTRAN using the message-passing model. In this model, parallel programs consist of a set of processes that communicate with each other by sending and receiving messages. Processes have separate address spaces and the programmer has explicit control over the memory used by each process and how the communication occurs.

MapReduce (MR) [7] is a programming model and associated distributed execution framework originally proposed for processing large datasets in commodity clusters. For many years it has been the de-facto standard for cloud programming. A program in MR is composed of a pair of user-provided map and reduce functions, generally written in Java, that are executed in parallel over a distributed network of worker processes. Executions can be described as a directed acyclic data flow where a network of stateless mappers and reducers process data in single batches, using a distributed filesystem (typically HDFS) to take the input and store the output. The execution framework has a master-worker architecture with implicit support for data distribution, parallel processing and fault tolerance.

Resilient Distributed Datasets (RDD) [8] is the main abstraction provided by Spark for supporting fault-tolerant and efficient distributed in-memory computations. An RDD is a read-only fault-tolerant partitioned collection of records that is created from other RDDs or from data in stable storage by means of coarse-grained transformations (e.g., map, filter or join) that apply the same operation to many data items at once. RDDs are used in actions (e.g. count, collect or save) to return a value to the application or export data to a storage system. Spark provides a programming interface to RDDs for Scala, Java or Python, and a distributed execution framework composed of a single driver program and multiple workers, that are long-lived processes that persist RDD partitions across operations. Developers write the driver program in which they can manipulate RDDs using a rich set of operators, control their partitioning to optimize data placement and explicitly persist intermediate results to memory or disk.

Each of the selected approaches has its pros and cons. The MPI approach is HPC oriented and its main advantage is the high communications performance combined with the use of a compiled programming language. The other two approaches are HTC oriented, having JVM-based distributed execution frameworks and using interpreted or JVM languages, which a priori will provide worst performance. On the other hand, compared with the other two, the MPI approach is too low level, requiring that the programmer deals with details like the data distribution, the ordering of communications or the fault-tolerance support.

Both MR and Spark provide high-level data abstractions, interfaces to object and/or functional languages and distributed execution frameworks with implicit support for data distribution and fault tolerance, so programming is easier and less error-prone. The main difference between them is how they support iterative algorithms. MR has been designed to execute programs in batches, taking input, and storing output and intermediate data in the file system. Executing iterative algorithms has considerable overhead because there is no way of efficiently reusing data or computation from previous batches. On the contrary, Spark has been designed with iterative algorithms in mind, providing efficient in-memory processing between iterations.

3 Parallel implementation of the Differential Evolution metaheuristic

For our study we have selected one representative population-based metaheuristic that will be used as a test case, the Differential Evolution (DE) [9]. DE is very popular and has been successfully applied to many problems [10]. It is an stochastic iterative method (see algorithm 1) that starting from an initial population matrix composed of NP D-dimensional solution vectors (individuals), tries to achieve the optimal solution iteratively using vector differences to create new candidate solutions. In each iteration, new individuals are generated by means of operations (crossover - CR; mutation - F) performed among individuals in the population matrix. New solutions will replace old ones when its fitness value was better. A population matrix with optimized individuals is obtained as output of the algorithm. The best of which is selected as the solution close to optimal for the objective function of the model.

Different models have been proposed for the parallelization of metaheuristics [11]. We have considered two of the most popular:

1. The *master-slave* model, that preserves the behaviour of the sequential algorithm parallelizing the inner-loop of the DE. In this model a master processor distributes computation operations between the slave processors.

2. The *island-based* model, that divides the population in subpopulations (*islands*) where the DE algorithm is executed isolated. To preserve diversity and avoid getting stuck in local optima sporadic individual exchanges are performed among islands.

Our first approach was to parallelize the DE using the master-slave model. But when we were implementing it with Spark, we found an issue with the parallelization of the generation of new individuals. In the implementation of this model with Spark, the population is partitioned and distributed among workers.

Algorithm 1 Differential Evolution algorithm

Input: A population matrix P of size D x NP
Output: A matrix P whose individuals were optimized

1: **repeat**
2: **for** each element i of the P matrix **do**
3: choose randomly different $r_1, r_2, r_3 \in [1, NP]$
4: choose randomly an integer $j_r \in [1, D]$
5: **for** $j \leftarrow 1, D$ **do**
6: choose a randomly real $r \in [0, 1]$
7: **if** $r \leq CR$ or $j = j_r$ **then**
8: $u_i^{G+1}(j) \leftarrow x_{r1}^G(j) + F \cdot (x_{r2}^G(j) - x_{r3}^G(j))$
9: **else**
10: $u_i^{G+1}(j) \leftarrow x_i^G(j)$
11: **end if**
12: **end for**
13: evaluate (u_i^{G+1})
14: **if** $fitness(u_i^{G+1}) < fitness(x_i^G)$ **then**
15: $x_i^{G+1} \leftarrow u_i^{G+1}$
16: **else**
17: $x_i^{G+1} \leftarrow x_i^G$
18: **end if**
19: **end for**
20: **until** Stop conditions

For the mutation of each individual, three random different individuals have to be selected from the whole population but, unlike MPI, communication among workers is not allowed in Spark. Each worker only have access to individuals of its partition and the driver is the only that has access to the complete population.

From the alternatives we tried to deal with this problem, broadcasting a read-only copy of the whole population to every worker in each iteration was the option that has shown best benchmarking results. But even so, the communications overhead introduced by this solution was unfeasible. So we concluded that the master-slave model did not fit well with the distributed programming model of Spark and decided to discard it in favour of the island-based model. A complete discussion of the implementation of the master-slave model and the other alternatives we tried to deal with this issue can be found in [12].

3.1 Island-based DE implementations

Dividing the population into subpopulations reduces interprocessor communications and leads to more loosely coupled parallel applications. Therefore, at least theoretically, the island model should be more suitable for implementing parallel metaheuristics with programming abstractions like MR or RDDs. In this section we explain the basics of the three island-based implementations of the DE that we have studied and compared.

Figure 1 shows the schematic diagram of the implementation with MPI of an asynchronous island-based DE (`asynPDE`) first presented in [13]. Each process ex-

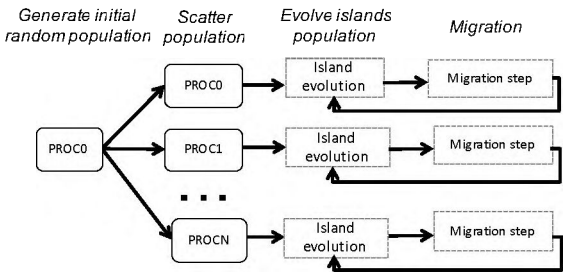

Figure 1: Island-based DE implementation with MPI

ecutes the same inner-loop of the sequential DE, doing mutation and crossover operations within the subpopulation of its island. A migration step is performed once every few iterations to migrate individuals among islands in order to preserve diversity. MPI asynchronous communications have been used to avoid having idle processes waiting for migrants arriving from other processes. The arriving of new individuals is examined at the end of the migration step (algorithm 2), proceeding with the next island evolution if new solutions have not yet arrived. Missed migrations will be checked again later, after each evolution of the island (algorithm 3). Asynchronous communications are also used for checking the stopping criteria of the algorithm. Whenever a process reaches a stopping condition, it sends a message to the rest so they can all stop almost at the same time.

As every MR program, our implementation of the island-based DE with MR (`mrPDE`) presented in [14] has two components: the main program (*driver*) that will be executed by the master and the *map* and *reduce* functions that will be executed by the workers. In the driver (algorithm 4) the outer-loop of the DE is executed to evolve a randomly-generated population until the stopping criterion is met. In every iteration, the population is partitioned into islands that are written to HDFS, a batch MR job is launched to evolve the islands in parallel and the resulting global population is read from HDFS once the job finished. To introduce diversity individuals are randomly shuffled during the partition of the population. The inner-loop of the DE algorithm is executed in the map function (algorithm 5) of the MR job. Each map instance reads an island population from HDFS and applies the mutation strategy during a predefined number of evolutions, taking random individuals from its island only. Finally, an output record is emitted for each individual of the evolved island using its fitness value as key. The MR job is completed with a single identity reducer that writes all the population to HDFS ordered by fitness.

The schematic diagram of our island-based DE implementation with Spark (`SiPDE`) presented in [12] is shown in figure 2. The population has been represented as a key-value pair RDD (solid outlined boxes in the figure). Each partition of the population RDD is considered to be an island (shaded rectangles in the figure, darker if they are persisted in memory). The algorithm starts generating in parallel an initial random population using a *map* transformation. Then, an *evolution-migration* loop is repeated until the stopping criterion, implemented as a *reduce* action (a distributed

Algorithm 2 Migration step of the MPI impl.

```
1: select(migSet);
2: // asynchronous send
3: ISend(migSet, remoteDest);
4: // asynchronous reception (non-blocking)
5: IRecv(recSet, remoteDest);
6: Test(recSet, isComplete);
7: if isComplete then
8:     Replace(recSet);
9: end if
```

Algorithm 3 Island evolution of the MPI impl.

```
1: repeat
2:     for each element in the population do
3:         cross(); mutation(); eval();
4:     end for
5:     while pending migration do
6:         // check pending messages (non-blocking)
7:         Test(recSet, isComplete);
8:         if isComplete then
9:             Replace(recSet);
10:        else
11:            break;
12:        end if
13:    end while
14: until Stop conditions
```

Algorithm 4 Driver of the MR implementation

Input: DE configuration parameters
Output: A population P of optimized individuals

```
1: P ← initial random population
2: #i ← number of islands
3: repeat
4:     // partition and shuffle the population
5:     Islands ⇐ PartitionPopulation(P, #i)
6:     P ⇐ EvolveIslands(Islands) // the MR job
7: until Stop conditions
```

Algorithm 5 Map function of the MR implementation

Input: An island I; DE configuration parameters
Output: An island I of optimized individuals

```
1: repeat
2:     // apply the DE mutation strategy
3:     I ⇐ EvolveIsland(I)
4: until number of evolutions
5: for each individual Ind of the island I do
6:     Emit(fitness(Ind), Ind)
7: end for
```

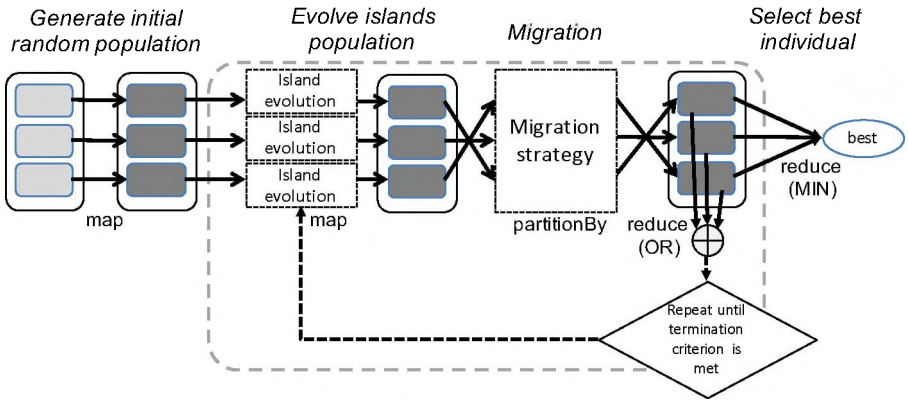

Figure 2: Island-based DE implementation with Spark

OR), is met. Finally, the best individual is selected using a *reduce* action (a distributed MIN). During each iteration of the *evolution-migration* loop islands evolve isolated for a predefined number of evolutions. After which, and to introduce diversity, the same random shuffling as in the MR implementation is performed using a custom random *partitioner*.

The three programming models have unique inherent features that have conditioned the implementations, modifying the systemic properties of the original algorithm, and resulting in different searches. As we will see later, this will influence the benchmarking results. The parts of the algorithm that have been implemented differently are:

- *The migration strategy*. While in MPI it is used an strategy with replacement (i.e. best individuals in one island replace worst individuals in the neighbour), in MR and Spark, as messaging between worker processes is not possible, the migration is a random shuffling of the population without replacement.

- *The evolution-migration synchronization*. In MPI individuals are migrated using asynchronous messaging between processes, so each island can proceed with a new evolution without waiting for the others. For the same reason as before, in MR and Spark that is not possible, so migrations introduce a synchronization step between island evolutions.

- *The stopping criterion checking*. In MPI whenever a process reaches a stopping condition sends an asynchronous message to the rest and all stop almost at the same time. Again that is not possible in MR and Spark, that have to wait until all the islands conclude their evolutions to check if any of them reached the stopping condition.

It must be noted that, although the migration steps in MR and Spark appear to be very similar, their implementations and overheads are very different. In MR the migration is performed in the *driver*, reading the population from HDFS, shuffling the individuals and writing back the islands to HDFS, while in Spark a custom random *partitioner* is used (no HDFS overhead).

4 Lessons learned from experimental results

In our previous works [14, 15, 16], we have conducted extensive testing of the three proposed island-based DE implementations using different benchmarks and execution platforms. In this section we overview the main lessons that we have learned, using the experimental results obtained for a challenging problem from the domain of computational systems biology, the *Circadian* model [17], as example. It is a parameter estimation problem in a nonlinear dynamic model of the circadian clock in the plant *Arabidopsis Thaliana*, that is known to be particularly hard due to its ill-conditioning and non-convexity.

Table 1 summarizes the results of the experiments with the MPI (`asynPDE`) and Spark (`SiPDE`) implementations. In the columns the number of cores (*#islands*) used, the mean number of evaluations required (*#evals*), the mean and deviation of the execution times (*time(s)*), and the speedup achieved versus the sequential execution are shown. Two different platforms have been tested varying the number of cores from 2 to 16/32 and calculating the mean of 20 independent runs for each configuration. The first platform was a cluster (named Pluton) with 16 nodes powered by two octa-core Intel Xeon E5-2660 @2.20GHz with 64GB and an InfiniBand FDR network. The second platform was a virtual cluster formed by A3 instances (4 cores, 7GB) in the Microsoft Azure cloud. The quality of the solution has been used as termination condition with a value-to-reach (*VTR*) of 1e-5 and, for the migration frequency, 50 evolutions between migrations have been used for `asynPDE` and 200 for `SiPDE`. Refer to [16] for the detailed experimental setting.

From table 1 we can conclude the following:

- *All the parallel implementations (including MR not shown in the table) have improved the conver-*

Table 1: MPI and Spark results (DE params: NP=256, D=13, CR=0.8, F=0.9, VTR=1e-5)

impl.	platform	#islands	#evals	time(s)	speedup
asynPDE	cluster	1	6,480,102	15,230.22±886.80	-
		2	3,540,889	4,078.36±1,852.32	3.73
		4	1,815,689	1,100.08±180.96	13.84
		8	1,231,094	380.99±77.64	39.97
		16	1,236,346	220.79±51.17	68.98
		32	1,700,782	149.82±30.37	101.65
	Azure	1	6,633,830	37,952.61±3,224.67	-
		2	3,067,622	9,196.63±1,110.82	4.13
		4	1,809,942	2,659.65±410.31	14.27
		8	1,279,609	929.77±204.21	40.82
		16	1,301,888	491.92±87.50	77.15
SiPDE	cluster	1	6,437,670	40883.39±3712.56	-
		2	5,980,416	19275.65±1281.63	2.12
		4	5,729,536	9305.30±909.41	4.39
		8	3,904,256	3319.33±296.88	12.32
		16	1,835,776	790.97±90.50	51.69
		32	1,577,216	348.36±43.47	117.36
	Azure	1	6,565,461	93,977.02±5216.28	-
		2	5,333,186	41,140.87±6474.26	2.28
		4	5,716,736	21,030.04±2443.06	4.47
		8	3,983,616	7,444.79±928.91	12.62
		16	1,953,536	1,768.25±166.51	53.15

gence time of the sequential algorithm. This was an expected result, given that, the evaluation of the population is performed in parallel. Moreover, superlinear speedups are obtained because, in the island model, the cooperation between islands improves the convergence rate.

- *The MPI implementation has the lower convergence rate (i.e. required number of evaluations), specially when using few cores*. This is due to the inherent features of each programming model explained in Section 3.1.

- *The convergence rate of the Spark implementation improves with the number of islands*, while the MPI implementation stagnates for more than 8. The reason is that the shuffling of the population, used as migration strategy in the Spark implementation, maintains the diversity when the number of islands increases.

- *The convergence rate has similar results in Azure, but execution time worsens and speedup improves*. The execution times are between 2x and 3x times worse in Azure than in the cluster due to the overhead of virtualization and using non-dedicated resources. By the contrary, the speedups are larger due to the computation-to-communication ratio, that is, the ratio between the time spent computing and communicating.

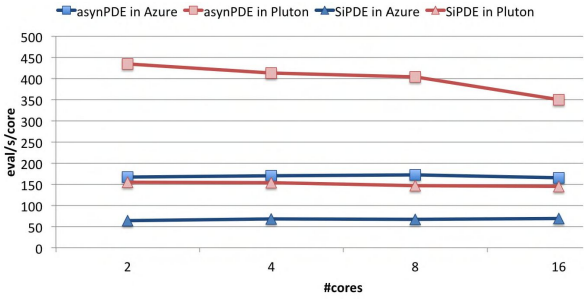

Figure 3: Evals/s/core of the MPI and Spark impl.

We have also calculated the number of evaluations per second and per core (*eval/s/core*) for both implementations and platforms (Figure 3). This is a good metric for evaluating the implementations because it includes not only the CPU time of evaluations, but also the communications and overhead time. From the figure we can conclude that:

- *The MPI implementation has a value of eval/s/core between 2.1x and 2.5x times better than the Spark implementation.*

- *The number of eval/s/core of the MPI implementation in the cluster drops with the growth of the number of cores*. This is because as the number of cores increases, so does the communications overhead. Besides, the computation of each is-

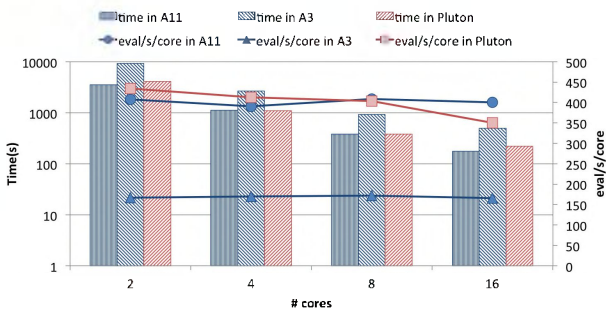

Figure 4: Comparison of MPI results in a physical cluster and in ordinary and HPC virtual clusters

land decreases, degrading the trade-off between computation and communication. Conversely, in the Spark implementation, the migration communication overhead is always the same, spreading the data movement among the available cores, and thus, having better scalability.

In order to evaluate the suitability of virtual clusters composed of HPC instances for running MPI applications, we have repeated the experiments with the MPI implementation in Azure, but now using a virtual cluster of A11 instances (16 cores, 112GB, Intel Xeon E5-2670 @2.6GHz). Figure 4 shows the results compared with those of table 1. From the figure we can conclude that:

- *The MPI implementation running in an HPC virtual cluster shows competitive execution times and better scalability.*

Figures 5 and 6 summarize the results of the experiments with the MR (`mrPDE`) and Spark (`SiPDE`) implementations in the cluster. Bean plots are used to show the execution times and the dispersion of the results. From the figures we can conclude that:

- *The MR implementation has larger execution times and dispersion than the Spark implementation.*
- *The MR implementation also reduces the convergence rate but with a limited speedup.*

To evaluate the overhead that is limiting the speedup of MR, we have measured, for a total of 8 iterations, the overhead of each *evolution-migration* iteration separately. These experiments were performed with versions of our implementations with the evolution of the population removed. Figure 7 shows the average and standard deviation of 10 independent runs of each experiment in two different platforms: the cluster we used before, and a virtual cluster formed by `m3.medium` instances (1 core, 3.75GB) in the AWS cloud. Refer to [14] for the detailed experimental setting. From the figure we can conclude that:

- *MR has significant higher overhead and larger dispersion than Spark.* It must be noted that,

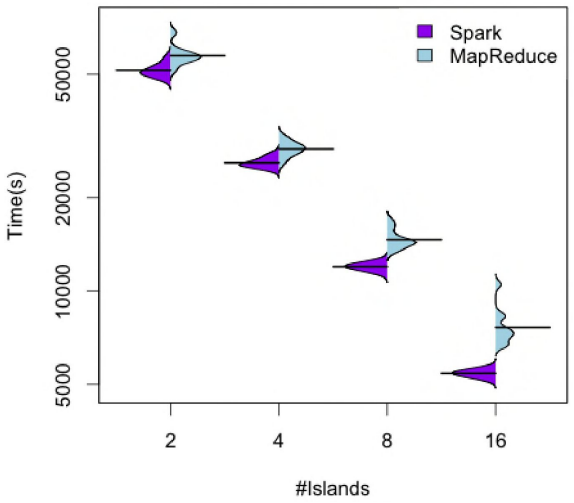

Figure 5: MR and Spark results in the cluster (DE params: NP=640, D=13, CR=0.8, F=0.9, VTR=1e-5)

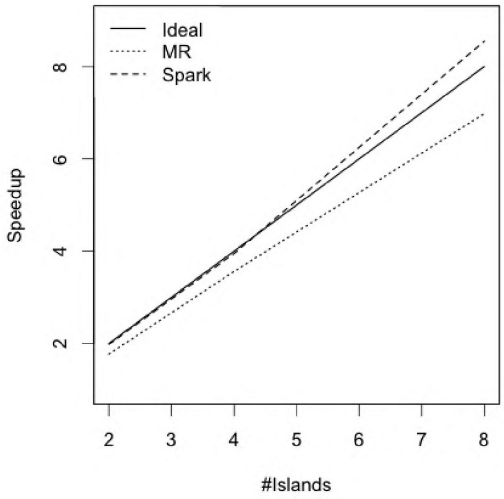

Figure 6: MR and Spark speedup in the cluster

although at first this would advise against using it in favour of Spark, for problems where the computation time significantly dominates over the overhead introduced by the iterations, MR would be competitive with Spark though with worst scalability.

- *Spark has better support for iterative algorithms than MR.* This was an expected result, given that, Spark persists results in-memory between iterations. That explains why the first iteration (the outliers in the box plots) is the most time consuming in Spark, while there is no significant difference between iterations in MR.
- *The overhead of MR and Spark is larger in AWS than in the cluster.* Spark is between 4x and 5x times worst in AWS, and MR around 2,8x. Also it must be noted that, in tune with what was expected, the Spark overhead slightly increases

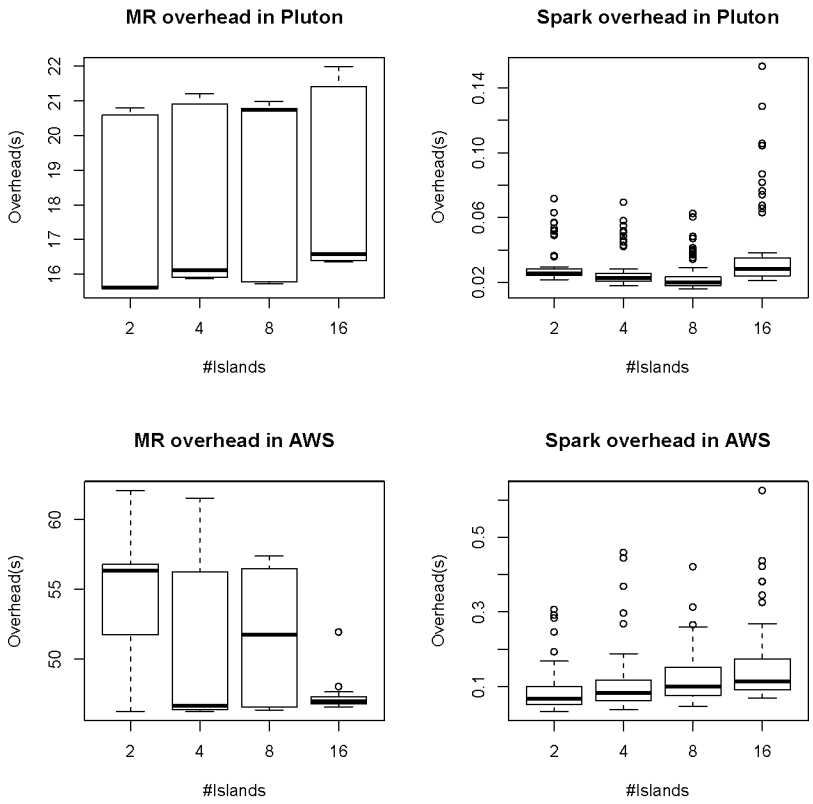

Figure 7: MR and Spark overhead per evolution-migration iteration

when the number of nodes grows.

5 Related work

Many works have been published analyzing the performance of HPC applications in the cloud. In [5, 6] extensive analysis to detect the more critical issues and bottlenecks are presented. These works conclude that the lack of high-bandwidth low-latency networks and the virtualization overhead has a great impact on the performance of tightly-coupled HPC workloads, as is the case of MPI applications.

There are also many works on the parallelization of metaheuristics. Nice reviews using traditional HPC approaches can be found in [11, 18]. With regard to the parallelization of DE, a distributed implementation using an island-model with asynchronous communications is proposed in [19].

Some works have studied the use of cloud programming models for the parallelization of metaheuristics. Most of these references are based on the use of MR, since not so long ago it was the de-facto standard for cloud programming. MRPSO, a parallelization of the Particle Swarm Optimization (PSO) metaheuristic is proposed in [20]. Different approaches to the implementation of parallel Genetic Algorithms (GA) are proposed in [21, 22]. In [23] different algorithmic patterns of distributed Simulated Annealing (SA) are designed and evaluated on Azure. In [24], a practical framework to infer large gene networks using a parallel hybrid GA-PSO is proposed.

There are also some references proposing parallelizations of the DE algorithm with MR. In [25] the fitness evaluation is performed in parallel, but experimental results show that the HDFS overhead reduces the benefits of the parallelization. In [26], a concurrent implementation of the DE steady-state model is proposed, but its applicability is limited to shared-memory architectures. In [27] it is proposed a parallel implementation of a DE-based clustering algorithm.

As Spark replaces MR as the new de-facto standard for cloud programming, the number of references using it is gradually increasing. A PSO proposal for data clustering in learning analytics can be found in [28]. PSO is also used in [29] to test the proposal of a parallel metaheuristic data clustering framework. In [30] a GA is parallelized for pairwise test generation. A MAX-MIN Ant System algorithm (MMAS) is presented in [31] to solve the Traveling Salesman Problem (TSP). A coral reef (CR) algorithm is applied to the job shop scheduling problem (JSP) in [32]. A Binary Cuckoo Search algorithm is applied to the crew scheduling problem (CrSP) in [33].

The first parallelization of the DE algorithm using Spark was proposed in [34]. However, only the fitness evaluation is performed in parallel following a

master-slave approach. An entire parallelization of the algorithm was first explored in [12]. In that paper we propose and evaluate implementations of the master-slave and the island-based models. Results showed that the island-based model is by far the best suited to be implemented using Spark. A thorough evaluation of the island-based implementation can be found in [15] and extensive comparatives of the implementations overviewed in this paper in [14, 16].

6 Conclusions

In this paper we overview the insights we have learned, through extensive experimentation, about some approaches to implement parallel metaheuristics in the cloud. Using the Differential Evolution metaheuristic and the Circadian model, a difficult parameter estimation problem from computational systems biology, as test case, three different programming paradigms: MPI, MapReduce and RDDs, have been evaluated on three different platforms: a cluster, the Azure cloud and the AWS cloud. Results show that, as it was expected, MPI outperforms the other approaches in terms of execution time, due to its reduced overhead and low level programming interface. Nevertheless, Spark should be positively consider when looking for better scalability, easier programmability and implicit support for data distribution and fault-tolerance. In our experience, to get efficient implementations using cloud programming models it is necessary to reshape the existing algorithms or to propose new ones.

The source code of the MPI and Spark implementations are publicly available at `https://bitbucket.org/xcpardo/sipde` and `https://bitbucket.org/pglez/asynpde`, respectively.

Acknowledgements

This research received financial support from the Spanish Government through the projects DPI2017-82896-C2-2-R, and TIN2016-75845-P (AEI/FEDER, UE), and from the Galician Government under the Consolidation Program of Competitive Research Units (Network Ref. R2016/045 and Project Ref. ED431C 2017/04), all of them co-funded by FEDER funds of the EU. We also acknowledge Microsoft Research for being awarded with a sponsored Azure account.

Competing interests

The authors have declared that no competing interests exist.

References

[1] C. A. Floudas and P. M. Pardalos, *Optimization in computational chemistry and molecular biology: local and global approaches*, vol. 40. Springer Science & Business Media, 2013.

[2] J. R. Banga, "Optimization in computational systems biology," *BMC systems biology*, vol. 2, no. 1, p. 47, 2008.

[3] I. E. Grossmann, *Global optimization in engineering design*, vol. 9. Springer Science & Business Media, 2013.

[4] I. Foster and D. B. Gannon, *Cloud Computing for Science And Engineering*. The MIT Press, 2017.

[5] R. R. Expósito, G. L. Taboada, S. Ramos, J. Touriño, and R. Doallo, "Performance analysis of HPC applications in the cloud," *Future Generation Computer Systems*, vol. 29, no. 1, pp. 218–229, 2013.

[6] I. Sadooghi, J. H. Martin, T. Li, K. Brandstatter, K. Maheshwari, T. P. P. de Lacerda Ruivo, G. Garzoglio, S. Timm, Y. Zhao, and I. Raicu, "Understanding the performance and potential of cloud computing for scientific applications," *IEEE Transactions on Cloud Computing*, vol. 5, no. 2, pp. 358–371, 2017.

[7] J. Dean and S. Ghemawat, "MapReduce: simplified data processing on large clusters," in *The 6th USENIX Symposium on Operating Systems Design and Implementation*, 2004.

[8] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2012.

[9] K. V. Price, R. M. Storn, and J. Lampinen, *Differential Evolution - a practical approach to global optimization*, vol. 141. 01 2005.

[10] S. Das and P. Suganthan, "Differential evolution: A survey of the state-of-the-art," *IEEE Transactions on Evolutionary Computation*, vol. 15, no. 1, pp. 4–31, 2011.

[11] E. Alba, G. Luque, and S. Nesmachnow, "Parallel metaheuristics: recent advances and new trends," *International Transactions in Operational Research*, vol. 20, no. 1, pp. 1–48, 2013.

[12] D. Teijeiro, X. C. Pardo, P. González, J. R. Banga, and R. Doallo, *Applications of Evolutionary Computation: 19th European Conference, EvoApplications 2016, Proceedings, Part II*, ch. Implementing Parallel Differential Evolution on Spark, pp. 75–90. 2016.

[13] D. R. Penas, J. R. Banga, P. González, and R. Doallo, "Enhanced parallel differential evolution algorithm for problems in computational systems biology," *Applied Soft Computing*, vol. 33, pp. 86 – 99, 2015.

[14] D. Teijeiro, X. C. Pardo, D. R. Penas, P. González, J. R. Banga, and R. Doallo, "Evaluation of parallel differential evolution implementations on mapreduce and spark," in *Euro-Par 2016: Parallel Processing Workshops*, pp. 397–408, Springer International Publishing, 2017.

[15] D. Teijeiro, X. C. Pardo, P. González, J. R. Banga, and R. Doallo, "Towards cloud-based parallel metaheuristics: A case study in computational biology with differential evolution and spark," *The International Journal of High Performance Computing Applications*, 2016.

[16] P. González, X. C. Pardo, D. R. Penas, D. Teijeiro, J. R. Banga, and R. Doallo, "Using the cloud for parameter estimation problems: Comparing spark vs mpi with a case-study," in *IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid '17)*, pp. 197–206, 2017.

[17] J. Locke, A. Millar, and M. Turner, "Modelling genetic networks with noisy and varied experimental data: the circadian clock in arabidopsis thaliana," *Journal of Theoretical Biology*, vol. 234, no. 3, pp. 383–393, 2005.

[18] T. G. Crainic, "Parallel meta-heuristics and cooperative search," *CIRRELT*, vol. 58, 2017.

[19] J. Apolloni, J. Garca-Nieto, E. Alba, and G. Leguizamn, "Empirical evaluation of distributed differential evolution on standard benchmarks," *Applied Mathematics and Computation*, vol. 236, pp. 351–366, 2014.

[20] A. W. McNabb, C. K. Monson, and K. D. Seppi, "Parallel PSO using MapReduce," in *IEEE Congress on Evolutionary Computation, CEC2007*, pp. 7–14, IEEE, 2007.

[21] C. Jin, C. Vecchiola, and R. Buyya, "MRPGA: an extension of MapReduce for parallelizing genetic algorithms," in *IEEE Fourth International Conference on eScience, eScience'08*, pp. 214–221, IEEE, 2008.

[22] A. Verma, X. Llora, D. E. Goldberg, and R. H. Campbell, "Scaling genetic algorithms using MapReduce," in *Ninth International Conference on Intelligent Systems Design and Applications, ISDA'09*, pp. 13–18, IEEE, 2009.

[23] A. Radenski, "Distributed simulated annealing with MapReduce," in *Applications of Evolutionary Computation*, pp. 466–476, Springer, 2012.

[24] W.-P. Lee, Y.-T. Hsiao, and W.-C. Hwang, "Designing a parallel evolutionary algorithm for inferring gene networks on the cloud computing environment," *BMC systems biology*, vol. 8, no. 1, p. 5, 2014.

[25] C. Zhou, "Fast parallelization of differential evolution algorithm using MapReduce," in *Proceedings of the 12th Annual Conference on Genetic and Evolutionary Computation*, pp. 1113–1114, ACM, 2010.

[26] K. Tagawa and T. Ishimizu, "Concurrent differential evolution based on MapReduce," *International Journal of Computers*, vol. 4, no. 4, pp. 161–168, 2010.

[27] M. Daoudi, S. Hamena, Z. Benmounah, and M. Batouche, "Parallel differential evolution clustering algorithm based on MapReduce," in *6th International Conference of Soft Computing and Pattern Recognition (SoCPaR)*, pp. 337–341, IEEE, 2014.

[28] K. Govindarajan, D. Boulanger, V. S. Kumar, and Kinshuk, "Parallel particle swarm optimization (ppso) clustering for learning analytics," in *2015 IEEE International Conference on Big Data (Big Data)*, pp. 1461–1465, Oct 2015.

[29] C.-W. Tsai, S.-J. Liu, and Y.-C. Wang, "A parallel metaheuristic data clustering framework for cloud," *Journal of Parallel and Distributed Computing*, vol. 116, pp. 39–49, 2018.

[30] R. Qi, Z. Wang, and S. Li, "Pairwise test generation based on parallel genetic algorithm with spark," in *Proceedings of the International Conference on Computer Information Systems and Industrial Applications*, pp. 67–70, 2015.

[31] L. Wang, Y. Wang, and Y. Xie, "Implementation of a parallel algorithm based on a spark cloud computing platform," *Algorithms*, vol. 8, no. 3, pp. 407–414, 2015.

[32] C.-W. Tsai, H.-C. Chang, K.-C. Hu, and M.-C. Chiang, "Parallel coral reef algorithm for solving jsp on spark," in *2016 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*, pp. 001872–001877, Oct 2016.

[33] J. Garcia Conejeros, F. Altimiras, A. Peña Fritz, G. Astorga, and O. Peredo, "A binary cuckoo search big data algorithm applied to large-scale crew scheduling problems," *Complexity*, 05 2018.

[34] C. Deng, X. Tan, X. Dong, and Y. Tan, "A parallel version of differential evolution based on resilient distributed datasets model," in *Bio-Inspired Computing-Theories and Applications*, pp. 84–93, Springer, 2015.