Using the Cloud for parameter estimation problems: comparing Spark vs MPI with a case-study

Patricia González*, Xoán C. Pardo*, David R. Penas†, Diego Teijeiro* and Julio R. Banga† and Ramón Doallo*
*Computer Architecture Group. Universidade da Coruña. Spain
Email:{patricia.gonzalez,xoan.pardo,diego.teijeiro,doallo@udc.es}
†BioProcess Engineering Group. IIM-CSIC. Spain
Email:{davidrodpenas,julio@iim.csic.es}

***Abstract*—Systems biology is an emerging approach focused in generating new knowledge about complex biological systems by combining experimental data with mathematical modeling and advanced computational techniques. Many problems in this field are extremely challenging and require substantial supercomputing resources to be solved. This is the case of parameter estimation in large-scale nonlinear dynamic systems biology models. Recently, Cloud Computing has emerged as a new paradigm for on-demand delivery of computing resources. However, scientific computing community has been quite hesitant in using the Cloud, simply because traditional programming models do not fit well with the new paradigm, and the earliest cloud programming models do not allow most scientific computations being efficiently run in the Cloud. In this paper we explore and compare two distributed computing models: the MPI (message-passing interface) model, that is high-performance oriented, and the Spark model, which is throughput oriented but outperforms other cloud programming solutions adding improved support for iterative algorithms through in-memory computing. The performance of a very well known metaheuristic, the Differential Evolution algorithm, has been thoroughly assessed using a challenging parameter estimation problem from the domain of computational systems biology. The experiments have been carried out both in a local cluster and in the Microsoft Azure public cloud, allowing performance and cost evaluation for both infrastructures.**

***Keywords*-Parallel Metaheuristics; Differential Evolution; Cloud Computing; MPI; Spark; Microsoft Azure**

I. Introduction

The aim of systems biology is to generate new knowledge about complex biological systems by combining experimental data with mathematical modeling and advanced computational techniques. Dynamic models (described by sets of nonlinear ordinary differential equations) is the most common formalism in this area. Model calibration consists of finding the parameters of a dynamic model that give the best fit to a set of time-series experimental data, which entails minimizing a cost function that measures the goodness of this fit. This class of problems is extremely challenging due to its ill-conditioned and multimodal nature [1]. To efficiently solve the calibration problem many research efforts have focused on developing metaheuristic methods, which combine mechanisms for exploring the search space and exploiting previous obtained knowledge, to find good solutions in reasonable computation times [2]. However, in most realistic applications, also the metaheuristic methods require a very large number of evaluations (and therefore, large computational time) to obtain an acceptable result. Thus, high-performance computing (HPC) solutions, that include parallel implementations and the use of HPC platforms, such as supercomputers or commodity clusters, have been extensively explored in the field. The parallelization of metaheuristics pursues one or more of the following goals: increase the size of the problems that can be solved, speed-up the computations, or attempt a more thorough exploration of the solution space. However, the success of the parallel solution, particularly when we are considering very challenging problems, is subject to be attended by the provision of a large number of resources, which is not always practicable.

With the advent of Cloud Computing effortless access to large number of distributed resources has become more feasible. However, its adoption by the HPC community has been limited. First because of the difficulty in employing cloud-based resources. The traditional parallel programming models and tools in HPC community are not easily applicable to the cloud platforms, and the learning curve to understand the different architectures and runtime environments of various cloud platforms discourage from adopting it as an alternative computational system. Second because clouds also raise important challenges regarding performance aspects. Recently, there have been many research works evaluating the promise of cloud platforms for HPC computing, most of them concluding that cloud-based clusters need a significant performance improvement to become competitive for HPC applications.

The main goal of this paper is to assess the performance of a well-known and popular metaheuristic for global optimization, the Differential Evolution (DE) algorithm, using different programming models: MPI (message-passing interface) which is the *de-facto* standard for HPC distributed computing, and Spark which is a cloud-oriented framework that outperforms other cloud solutions for iterative algorithms. Besides, the results obtained in different infrastructures, namely a local cluster, which is HPC-oriented, and the

Microsoft Azure public cloud, which is throughput oriented, have been thoroughly discussed. A very challenging parameter estimation problem from the domain of computational systems biology has been used as benchmark. However, the results obtained in this paper can be particularly useful, not only for the computational systems biology community, but also for those interested in the potential of cloud frameworks and platforms for developing metaheuristic methods in global optimization problems in general.

The structure of this paper is as follows. Section II discusses related work and briefly describes the two programming models evaluated here, MPI and Spark, focusing on its suitability for the cloud. Section III introduces the parallel implementations of the DE used in this paper. Section IV describes the experiments carried out and discusses on the obtained results. Finally, Section V summarizes the conclusions of the paper.

II. Background and related work

A. Parallel programming models and frameworks

Computational science has appeared as a new discipline several decades ago stimulated by the rapid increase of computational power. Its success has caused demand for supercomputing resources to rise sharply, and parallel computers have evolved to become the everyday tool for computer scientists. Message passing is by far the most widely used approach to parallel computing, specially on large parallel systems, typically with distributed memory. The message-passing model consists of a set of processes that are able to communicate with each others by sending and receiving messages. In the message-passing model of parallel computation, the processes executing in parallel have separate address spaces. Communication occurs when a portion of one process's address space is copied, in a cooperative way, into another process's address space. This model provides the programmer with explicit control over the location of memory in a parallel program, specifically, the memory used by each process. This ability to manage memory location can allow the programmer to achieve high performance. However, the main drawback of message passing is that the programmer needs to pay attention to details such as the placement of memory and the ordering of communications. The Message-Passing Interface (MPI) is a library that allows developers to write robust and efficient parallel and distributed applications using the message-passing paradigm. MPI is, probably, the most widely used programming framework in the HPC community. In the last decade, several researchers have studied the performance of MPI applications in the cloud. Most of these studies use classical MPI benchmarks to compare the performance of MPI on public cloud platforms. The NAS benchmarks have been used in [3], while the Linpack benchmark has been employed in [4]. In [5] a variety of microbenchmarks and kernels were studied. Besides, also real applications have been assessed in the cloud, such as bioinformatics applications [6], high-energy and nuclear physics experiments [7], and different e-Science applications [8], [9]. In [10] the performance of a set of applications that represent the typical workload run at a supercomputing center have been examined. Also, an extensive analysis to detect the more critical issues and bottlenecks of HPC applications in the cloud has been carried out in [11]. These works conclude that clouds were not designed for running tightly-coupled HPC workloads, like MPI applications. Overall, the lack of high-bandwidth, low-latency networks, as well as the virtualization overhead, has a large effect on the performance of such applications in the Cloud.

From the new programming models that have been proposed to deal with large scale computations on cloud systems, MapReduce [12] 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.

However, when it comes to iterative algorithms MapReduce has shown serious performance bottlenecks [13] mainly because there is no way of reusing data or computation from previous iterations efficiently. New proposals, not based on MapReduce, like Spark [14] or Flink, which has its roots on Stratosphere [15], 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 [14] the performance of iterative algorithms can be improved by an order of magnitude when compared to MapReduce.

Additionally, in an attempt of converging cloud platforms and HPC, projects like DataMPI [16] or CloudMPI [17] arose. DataMPI [16] aimed at extending MPI by key-value pair based communication operations to provide high performance communications in cloud scenarios. The cloudMPI [17] framework aimed at designing and implementing an MPI-like framework for cloud platforms, having an early implementation on the Azure cloud platform. Unfortunately, none of these projects seems to be active at this moment.

B. Parallel metaheuristics in the cloud

The parallelization of metaheuristics methods has received much attention to reduce the time for solving large-scale problems [18]. Many parallel algorithms have been proposed in the literature, some of them focussed on the parallelization of DE. Nice reviews can be found in [19], and more recently in [20]. Most of these proposals are parallel implementations based on traditional parallel programming interfaces. Research on cloud-oriented parallel metaheuristics based mainly on the use of MapReduce has also received increasing attention in recent years [21], [22], [23], [24], [25]. Some proposals are specific on studying how to apply MapReduce to parallelize the DE algorithm to be used in the Cloud. In [26] 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. The use of Spark for the parallelization of the DE algorithm was explored in [27]. In this paper Spark-based implementations of two different parallel schemes of the DE algorithm, the master-slave and the island-based, were proposed and evaluated. Results showed that the island-based solution is by far the best suited to the distributed nature of Spark. Also, a comparison of the previous Spark implementation of the DE algorithm with a MapReduce implementation has been performed in [28], already concluding that Spark outperforms MapReduce in this kind of iterative algorithms.

Note that, although there are many studies in the literature evaluating the performance of a specific programming model or framework in different computing platforms using traditional benchmarks, we think that there is a lack of research on assessing and discussing the performance of a particular kind of widely used applications using different models and platforms. Recently, in [29] the trade-offs of performing linear algebra, specifically matrix factorizations, using Spark were compared to tradicional MPI implementations. The head-to-head comparisons of those Spark and MPI implementations revealed a number of opportunities for improving Spark performance. Those experiments were performed in a supercomputer. In this paper we will explore the implications of the use of MPI and Spark in the parallel implementation of the DE algorithm. We will discuss about the differences that arise from the inherent features of each programming model, and we will assess the performance of both implementations in different computing platforms, a local cluster and the Microsoft Azure public cloud.

III. Parallel Differential Evolution implementations

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. From now on we will describe and use the DE basic algorithm reported in [30], specifically the DE/rand/1 scheme, however, the parallel implementations described here can be both applied to other DE schemes. Algorithm 1 shows the basic pseudocode for the specific version of the DE algorithm used in this paper. For each iteration, new individuals are generated in the population matrix through operations (crossover - CR; mutation - F) performed among individuals of the matrix, with old solutions replaced 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 is selected as solution close to optimal for the objective function of the model.

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. In the literature, different parallel models can be found [18] aiming to improve both computational time and number of iterations for convergence. The *master-slave* 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. A master processor distributes computation operations between the slave processors. Therefore, the parallel algorithm has the same behavior of the sequential one.

The parallel implementations evaluated in this paper are both based on the island model approach. The population

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 i of the P matrix do
        choose randomly different r1, r2, r3 ∈ [1, NP]
        choose randomly an integer j_r ∈ [1, D)
        for j ← 1 to D do
            choose a randomly real r ∈ [0, 1]
            if r ≤ CR or j = j_r then
                u_i^{G+1}(j) ← x_{r1}^G(j) + F · (x_{r2}^G(j) − x_{r3}^G(j))
            else
                u_i^{G+1}(j) ← x_i^G(j)
            end
        end
        evaluate (u_i^{G+1})
        if f(u_i^{G+1}) < f(x_i^G) then
            x_i^{G+1} ← u_i^{G+1}
        else
            x_i^{G+1} ← x_i^G
        end
    end
until Stop conditions;
```

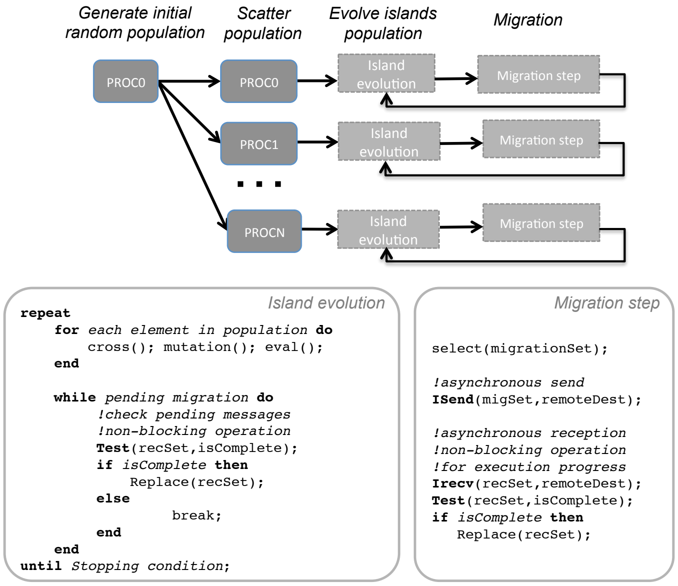

Figure 1: MPI asynchronous island-based implementation of the DE algorithm (`asynPDE`).

matrix is divided in subpopulations (*islands*) where the algorithm is executed isolated. Phases such as selection, recombination and mutation are performed only within each island, which implies absence of collaboration among processes. Sparse individual exchanges are performed among islands to introduce diversity into the subpopulations, preventing search from getting stuck in local optima.

A. MPI asynchronous Parallel DE

The simplest implementation of the parallel island DE is a synchronous algorithm. The drawback of the synchronous algorithm is that processors are idle during a significant amount of time, while they are waiting for each other during the migration steps. Replacing synchronous communications with asynchronous ones, each process will send the information to a memory buffer associated with the remote process, enabling the reception of the message later on (whenever that process is ready to receive it), thus, avoiding idle periods.

For the evaluation carried out in this work we have used the implementation of the asynchronous parallel DE (`asynPDE`) described in [31]. Figure 1 schematically illustrates this implementation. Each process receives an island population. For each iteration of the main loop, mutation and crossover operations are performed within each island, in the same way as in the serial implementation. Every m iterations, a migration phase is performed to link the evaluations in different islands. Whenever a process reaches the migration phase, it sends a set of individuals to the selected remote process using an asynchronous communication (`ISend()` function in MPI). Then, the process in the migration phase checks if the message with the new individuals of a remote process has already arrived to its memory buffer (using a `IRecv()` function). However, if the new solutions have not arrived yet, the process proceed with the next evaluation. After each iteration of the algorithm the process searches for the reception of data missed in previous migrations (`Test()` function), avoiding stalls if the messages have not arrived yet.

In addition to the migration step, the checking of the stopping criteria may also involve communications between processes. Stopping criteria are needed to terminate the execution of the algorithms. They can be as simple as using a maximum number of evaluations, which do not imply exchange of communications. However, other criteria, that allow to react adaptively to the state of the optimization progress, need communications between processes. Asynchronous MPI communications are also used in the proposed algorithm for those communications, so that processes may continue running independently. Each parallel process opens a buffer where it expects to receive a termination message. This buffer is checked every iteration of the algorithm. Thus, the control of the stopping criteria of the global search is distributed among all the processes: when a stopping condition is fulfilled in a process, this condition is communicated to the rest of processes, then all of them can stop the algorithm almost at the same time.

B. Spark island-based Parallel DE

To understand the Spark-based parallel implementation of the DE algorithm, some previous insight into the way data is distributed and processed by Spark is needed. Spark uses the *resilient distributed dataset* (RDD) abstraction to represent fault-tolerant distributed data. RDDs are immutable sets of records that optionally can be in the form of key-value pairs. Spark programs are run by a driver (the master in Spark terminology) which partitions RDDs and distributes the partitions to workers (the slaves in Spark terminology), that 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.

With the aim of better understanding Spark intricacies and assess the performance of different alternatives when implementing DE, in [27] we have presented a preliminary evaluation of different variants of the master-slave parallel implementation (`SmsPDE`), and an island-based parallel implementation (`SiPDE`). The main conclusion of that work is that the island-based parallel implementation is the best suited to the distributed nature of Spark and obtains the best performance results.

The Spark island-based parallel DE implementation (`SiPDE`) used in the evaluation performed in this paper

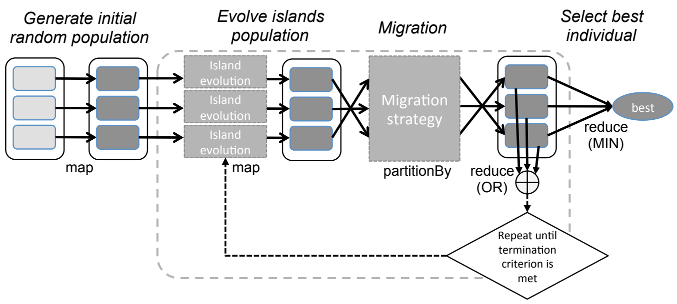

Figure 2: Spark island-based implementation of the DE algorithm (`SiPDE`).

follows the scheme shown in Figure 2. Boxes with solid outlines are RDDs. Partitions are shaded rectangles, darker if they are persistent in memory. A key-value pair RDD has been used to represent the population where each individual is uniquely identified by its key. Some steps in the main flow of the algorithm are executed in a distributed fashion:

- The random generation and initial evaluation of individuals that form the population, implemented as a Spark map transformation.
- The evolution of the population. Every partition of the population RDD is 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.
- The migration strategy, which introduces diversity by exchanging selected individuals among islands every time the evolution of the islands ends.
- The checking of the termination criterion, implemented as a Spark reduce action (a distributed OR operation).

The evolution-migration loop is repeated until the termination criterion is met, after which the best individual is selected by means of a Spark reduce action (a distributed MIN operation).

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 key hash. 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.

C. Key differences between the MPI and the Spark implementations

Three are the main differences between the MPI and the Spark implementations described above. All these differences arise from the inherent features of the programming model used in each implementation, and more specifically from the fact that the communication among workers is not allowed in Spark.

- *Migration strategy*. While in `asynPDE` the migration strategy consists of a selection of the best individuals in one island to replace the worst individuals in the neighbor, the migration strategy in `SiPDE` consists of randomly and evenly shuffling elements among islands without replacement.
- *Synchronization*. The use of a partitioner to perform the migration strategy leads to a synchronization step in the Spark implementation. The MPI implementation, on the contrary, performs the information exchange between islands through non-blocking asynchronous message-passing operations.
- *Stopping criterion checking*. Although the stopping criterion is evaluated during each island evolution, when it is met by one or more islands the Spark implementation only stops after the reduce operation at the end of the stage (see Figure 2). Thus, the Spark implementation cannot stop just right when the value to reach is attained in one of the islands, as `asynPDE` implementation does.

IV. Experimental Results

In order to carry out the proposed performance evaluation, a challenging parameter estimation problem from the domain of computational systems biology was considered, the *Circadian* model. It consists of a parameter estimation in a nonlinear dynamic model of the circadian clock in the plant *Arabidopsis thaliana*, as presented in [32]. The model contains 7 ordinary differential equations with 27 parameters (13 of them were estimated) with data sets from 2 experiments. This problem is known to be particularly hard due to its ill-conditioning and non-convexity [33], [1].

For the experimental testbed two different platforms have been used. First, experiments were conducted in our local cluster Pluton, that consists of 16 nodes powered by two octa-core Intel Xeon E5-2660 @2.20GHz CPUs with 64 GB of RAM, and connected through an InfiniBand FDR network. Second, experiments were deployed with default settings in the Microsoft Azure public cloud using clusters with A3 instances (4 cores, 7GB). For the MPI experiments a custom cluster with canonical Ubuntu Server nodes was used, while for the Spark experiments we used a standard HDInsight Spark cluster with A3 instances for head and worker nodes. All the nodes were located in the North Europe region, although there is no further guarantee on proximity of nodes allocated together, which can lead to significant variability in latency between nodes. Additionally, we had no control over in which underlying hardware the clusters were instantiated on. To avoid ending up with different virtual clusters in every experiment, we instantiated one cluster for MPI experiments and another cluster for the Spark experiments and all the tests have been performed in these clusters. By examining /proc/cpuinfo we have identified that the actual CPU used in both clusters was an Intel Xeon E5-2673 @2.40GHz with 7GB of RAM.

Comparing the different implementations of the parallel metaheuristic is not an easy task due to their key differences (see III-C) that affect the convergence rate of the algorithms. For this experimental evaluation, both the MPI and the Spark implementations were executed until their achieved solutions had a similar accuracy, *i.e.*, they are compared based on a quality solution. The target value, or value-to-reach (VTR), used as stopping criterion in the following experiments was 1.0e-5.

Additionally, there are many configurable parameters in the classical DE algorithm, such as the population size (NP), 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. The objective of this work is not to evaluate the impact of these parameters, thus, only results for one configuration are reported here. Previous tests have been done to select those parameters that lead to reasonable computation times. For the selection of the settings in these experiments, the suggestions in [30] have been followed. For all the experiments in this section NP=256, F=0.9, CR=0.8 and MSt=DE/rand/1 were used.

Besides, in parallel island DE algorithms, new parameters have to be also considered, such as the migration frequency (μ), the island size (λ), the communication topology between processes, or the selection and replacement policy in the migration step. The migration frequency will have a significant impact in the performance of both implementations, because a high migration rate will emphasize the communications overhead, particularly affecting the performance on cloud platforms. Thus, we performed a preliminary study to determine the optimal migration frequency for both implementations. Frequencies of 50 iterations between migrations for `asynPDE`, and of 200 iterations between migrations for `SiPDE`, have been chosen. In addition the island size will be $\lambda = NP/\#cores$. Finally, in the MPI implementation, the communication topology used is a star, and the selection policy consists in selecting only the best individual in the island population to be sent as a promising solution, while the replacement policy consists in replacing the worst individual in the island population with the incoming solution. Note that in the Spark implementation the migration step consists in a shuffle of the island populations instead of a selective send and replacement in each island.

Results for both `asynPDE` and `SiPDE` implementations in our local cluster Pluton are shown in Table I. We carried out experiments varying the number of cores, from 2 to 32. We have not used more than 32 cores because the scalability of the parallel DE algorithm is heavily restricted by the population size, and, according to the guideline of [30], the population size should be around 10D (being D the dimension of the problem, which in the case of the Circadian benchmark is 13). We have set the population size to 256 individuals, so as to be able to scale up to 32 islands of $\lambda = 8$ individuals each. And, although some authors [34] have shown that a population size lower than the dimensionality of the problem can be optimal in many cases, the fact is that the smaller the island population size is, the less chances for the combination between individuals and a lower convergence rate will be achieved. Table I displays, for each experiment, the number of cores (*#islands/cores*) 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. Results show that the parallelization improves the execution time required for convergence not only by performing the evaluations in parallel but also because the cooperation between islands leads to an improvement in the convergence rate (less evaluations are needed), thus, achieving superlinear speedups. As it can be seen, the number of required evaluations is lower in the MPI than in the Spark implementation, specially when the number of cores is also low. Besides the other differences between both implementations remarked in Section III-C that would significantly affect the convergence rate, it is worth noting that the checking of the stopping criterion in the Spark implementation results in a larger number of evaluations. The reason being that all the islands have to reach the reduce operation at the end of the evolution stage. Additionally, the convergence rate in both implementations differs when the number of islands increases. Although for a small number of islands the MPI implementation clearly outperforms in number of evaluations the `SiPDE` implementation, it should be noted that the convergence of the `asynPDE` implementation stagnates for more than 8 processes, while it improves for the Spark implementation. This can be also observed in the speedup trend, shown graphically in Figure 3. This important feature is due to difference in the migration strategy followed by both

Table I: Performance evaluation of `asynPDE` and `SiPDE` in local cluster Pluton.

meth.	#islands/cores	#evals	time(s)	speedup
asynPDE	1	6,480,102	15230.22±886.80	-
	2	3,540,889	4078.36±1852.32	3.73
	4	1,815,689	1100.08±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
SiPDE	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

Table II: Performance evaluation of `asynPDE` and `SiPDE` in Azure public cloud.

meth.	#islands/cores	#evals	time(s)	speedup
asynPDE	1	6,633,830	37952.61±3224.67	-
	2	3,067,622	9196.63±1110.82	4.13
	4	1,809,942	2659.65±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	1	6,565,461	93977.02±5216.28	-
	2	5,333,186	41140.87±6474.26	2.28
	4	5,716,736	21030.04±2443.06	4.47
	8	3,983,616	7444.79±928.91	12.62
	16	1,953,536	1768.25±166.51	53.15

implementations. When the number of islands increases, selecting the best individuals to be shared in the migration step leads to small populations full of cooperative solutions that has an adverse impact on diversity, and could even cause premature convergence to local optima. Shuffling the islands population, on the contrary, maintains the diversity of the searches when the number of islands increases.

The same experiments were performed in the Azure public cloud from 2 to 16 cores. Results are shown in Table II. As it can be seen, the Azure experiments obtain similar results as those of the local cluster in terms of convergence (number of evaluations required). However, results in terms of execution times and speedup differ. On the one hand, the overhead introduced in Azure due to virtualization and use of non-dedicated resources in a multitenant platform are not negligible, being the execution times obtained in Azure between 2x and 3x the times obtained in Pluton. On the other hand, the speedups, in particular when the number of cores grows, achieved in Azure are larger than in Pluton. This is due to the computation-to-communication ratio, that is, the ratio of the time spent computing to the time spent communicating, which depends on the relative speeds of the processor and the communication medium. In particular, for the `asynPDE` implementation the number of communications increases with the number of cores, and the computation, on its turn, decreases. Thus, since the computation is slower in Azure, the scalability is better in this platform.

To further compare the performance of both implementations without attending to the convergence rate achieved in each case, the number of evaluations per second and per core (eval/s/core) has been calculated. Note that this computation includes not only the CPU time for the evaluation itself but also the communication time and other overhead introduced by the algorithm implementation, thus, it is a good metric to assess the performance of Spark versus MPI for this problem. Figure 4 shows the eval/s/core achieved for both implementations and the two testbeds used. We encountered that the eval/s/core of the MPI implementation was between 2.1x and 2.5x the one obtained by the Spark implementation. It can also be observed the drop in the number of evaluations per second and core in Pluton for the `asynPDE` implementation when the number of cores grows. The reason is that the number of communications, and thus their overhead, increases with the number of cores in the MPI implementation, and, additionally, since the computation of each island decreases with the number of cores, the trade-off between computations and communications degrades. However, in the Spark implementation the number of communications in each shuffle is always the same, and this data movement is spread among the number of cores, thus, providing a good scalability.

Before finishing the evaluation we wondered how much performance we can obtain when using larger (and more expensive) instances in the Azure platform. Azure provides a number of different instance types that have varying performance characteristics and prices. Previous results were obtained in clusters of A3 instances. For this new comparison, we have also built a cluster with compute-intensive instances and we have carried out there the tests with the MPI implementation. We have chosen A11 compute-intensive instances that are 16-core nodes with Intel Xeon E5-2670 @2.6GHz CPUs with 112GB of RAM. Figure 5 shows the

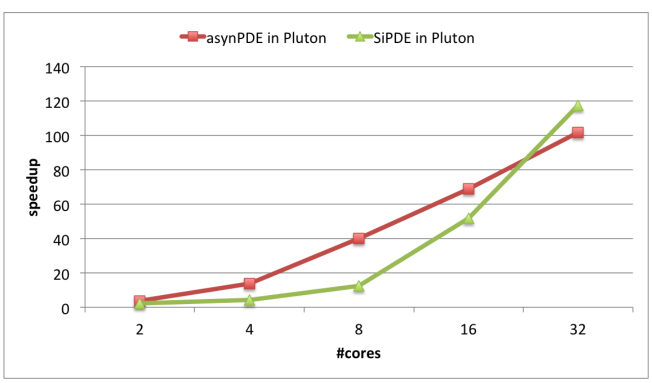

Figure 3: Speedup achieved by `asynPDE` vs `SiPDE` in Pluton.

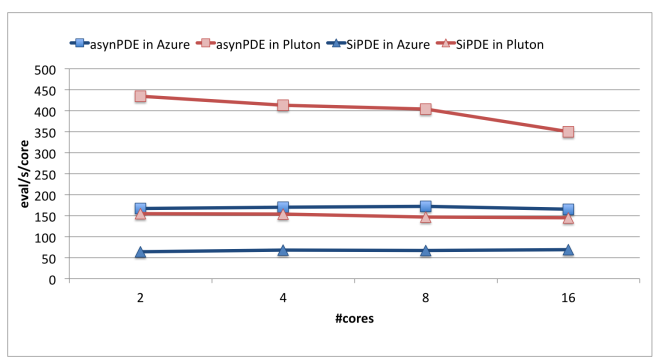

Figure 4: Evals/s/core achieved by `asynPDE` vs `SiPDE` in Pluton local cluster and Microsoft Azure public cloud.

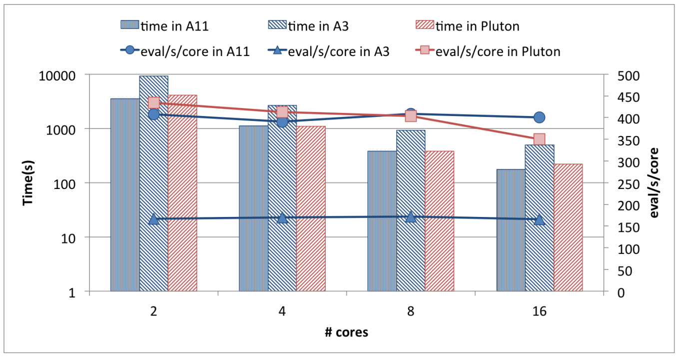

Figure 5: Comparison of `asynPDE` results in Pluton local cluster with results in a cluster of A3 instances and a cluster of A11 instances in Azure.

results obtained and its comparison with the results obtained in the previous Azure A3-cluster and also in the local cluster Pluton. It can be seen that execution times (shown in a logarithmic scale in the primary axis) in the A11-cluster are competitive with those obtained in the local cluster, and even outperforms Pluton when the number of cores grows, showing a better scalability. The number of evaluations per second and core is also shown in the secondary axis and clearly illustrate the improvement in scalability when using the compute-intensive Azure instances.

To conclude this evaluation we have found it interesting to carry out a brief study on the cost of these experiments in the Azure public cloud. Conducting a cost analysis comparing the cost of relying on cloud computing and that of owning an in-house cluster would be of particular interest, although it is a very difficult task [35]. The acquisition and operational expenses have to be used in estimating the local clusters cost. However, the actual cost of local clusters is related to its utilization level. For a local cluster acquired as one unit and maintained for several years, the higher the actual utilization level, the lower the effective cost rate. Besides, labor cost in management and maintenance should also be included, which could be significant. Thus, we found unfeasible an accurate estimation of the cost per hour in our local cluster. Besides, if we take a look to the price of the used instances, we can see that in February 2017 the cost of each A3-instance was 0.1585 €/hour, while the cost of the A11-instance was 1.3155 €/hour. To run the experiments in this paper we used 16-core clusters, so we needed four A3 instances but only one A11 instance. The mean pricing for each experiment is shown in Table III. Having into account that the performance, in terms of execution time, of the A11-cluster in the tests performed in this work has been of 2.5x over the A3-cluster, the use of A11 instances can be cost-effective. In view of the obtained results we can conclude that, though our experiments in the cloud demonstrates a slightly poorer performance compared to the local cluster, the cloud *pay-as-you-go* model can be potentially a cost-effective and timely solution for the needs of many users.

Table III: Cost evaluation in Azure public cloud for `asynPDE` experiments.

#cores	#instances A3	#instances A11	mean prize per run A3	mean prize per run A11
1	1	1	1.67 €	5.57 €
2	1	1	0.40 €	1.29 €
4	1	1	0.12 €	0.41 €
8	2	1	0.08 €	0.14 €
16	4	1	0.04 €	0.06 €

V. Conclusions

In this paper, we explore and compare the performance of a parameter estimation problem in computational systems biology using the well-known Differential Evolution algorithm implemented using two different distributed computing models: MPI, that is HPC oriented, and Spark which is throughput oriented. We have assessed both implementations in two different infrastructures: a local cluster and the Microsoft Azure public cloud. Results show that, as it was expected, from a computational point of view the MPI implementation outperforms the Spark one in terms of execution time. This is mainly due to its low level programming language and reduced overhead. Nevertheless, the Spark implementation should be positively considered since it allow easier programmability and because it also presents further advantages, such as inherent support to node failures and data replication.

Although this research was designed and tested with focus on the field of parameter estimation problems in computational systems biology, we believe that the results obtained in this work can be useful for those researchers interested in the performance of existing traditional parallel metaheuristics in new cloud platforms, as well as in those interested in the potential of new programming models for developing parallel metaheuristic methods.

The source code of the two parallel DE implementations used as benchmarks in this work, `SiPDE` and `asynPDE`, are publicly available at `https://bitbucket.org/xcpardo/sipde` and at `https://bitbucket.org/pglez/asynpde`, respectively.

Acknowledgment

This research received financial support from the Spanish Government through the projects DPI2014-55276-C5-2-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. GRC2013/055), all of them co-funded by

FEDER funds of the EU. We also acknowledge Microsoft Research for being awarded with a sponsored Azure account.

References

[1] A. Villaverde and J. R. Banga, "Reverse engineering and identification in systems biology: Strategies, perspectives and challenges," *Journal of the Royal Society Interface*, vol. 11, no. 91, p. art. no. 20130505, 2014.

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

[3] C. Evangelinos and C. Hill, "Cloud computing for parallel scientific HPC applications: Feasibility of running coupled atmosphere-ocean climate models on amazon's EC2," in *1st Workshop on Cloud Computing and its Applications (CCA'08)*, 2008, pp. 1–6.

[4] J. Napper and P. Bientinesi, "Can cloud computing reach the top500?" in *Proceedings of the combined workshops on UnConventional high performance computing workshop plus memory access workshop*. ACM, 2009, pp. 17–20.

[5] S. Ostermann, A. Iosup, N. Yigitbasi, R. Prodan, T. Fahringer, and D. Epema, "An early performance analysis of cloud computing services for scientific computing," *Delft University of Technology, Tech. Rep*, 2008.

[6] S. Hazelhurst, "Scientific computing using virtual high-performance computing: a case study using the Amazon elastic computing cloud," in *Proceedings of the 2008 annual research conference of the South African Institute of Computer Scientists and Information Technologists on IT research in developing countries: riding the wave of technology*. ACM, 2008, pp. 94–103.

[7] K. Keahey, T. Freeman, J. Lauret, and D. Olson, "Virtual workspaces for scientific applications," *Journal of Physics: Conference Series*, vol. 78, no. 1, p. 012038, 2007.

[8] L. Ramakrishnan, K. R. Jackson, S. Canon, S. Cholia, and J. Shalf, "Defining future platform requirements for e-Science clouds," in *Proceedings of the 1st ACM symposium on Cloud computing*. ACM, 2010, pp. 101–106.

[9] J. Li, M. Humphrey, C. Van Ingen, D. Agarwal, K. Jackson, and Y. Ryu, "e-Science in the cloud: A modis satellite data reprojection and reduction pipeline in the Windows Azure platform," in *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on*. IEEE, 2010, pp. 1–10.

[10] K. R. Jackson, L. Ramakrishnan, K. Muriki, S. Canon, S. Cholia, J. Shalf, H. J. Wasserman, and N. J. Wright, "Performance analysis of high performance computing applications on the amazon web services cloud," in *Cloud Computing Technology and Science (CloudCom), 2010 IEEE Second International Conference on*. IEEE, 2010, pp. 159–168.

[11] 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.

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

[13] J. Ekanayake, H. Li, B. Zhang, T. Gunarathne, S. hee Bae, J. Qiu, and G. Fox, "Twister: a runtime for iterative MapReduce," in *The First International Workshop on MapReduce and its Applications*, 2010.

[14] M. Zaharia and 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.

[15] A. Alexandrov, R. Bergmann, S. Ewen, J.-C. Freytag, F. Hueske, A. Heise, O. Kao, M. Leich, U. Leser, V. Markl, F. Naumann, M. Peters, A. Rheinlnder, M. Sax, S. Schelter, M. Hger, K. Tzoumas, and D. Warneke, "The stratosphere platform for big data analytics," *The VLDB Journal*, vol. 23, no. 6, pp. 939–964, 2014. [Online]. Available: http://dx.doi.org/10.1007/s00778-014-0357-y

[16] X. Lu, F. Liang, B. Wang, L. Zha, and Z. Xu, "DataMPI: extending MPI to Hadoop-like big data computing," in *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*. IEEE, 2014, pp. 829–838.

[17] D. Agarwal, S. Karamati, S. Puri, and S. K. Prasad, "Towards an MPI-like framework for the Azure cloud platform," in *Cluster, Cloud and Grid Computing (CCGrid), 2014 14th IEEE/ACM International Symposium on*. IEEE, 2014, pp. 176–185.

[18] 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.

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

[20] S. Das, S. S. Mullick, and P. Suganthan, "Recent advances in differential evolution–an updated survey," *Swarm and Evolutionary Computation*, vol. 27, pp. 1–30, 2016.

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

[22] 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*. IEEE, 2008, pp. 214–221.

[23] 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*. IEEE, 2009, pp. 13–18.

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

[25] 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.

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

[27] D. Teijeiro, X. C. Pardo, P. González, J. R. Banga, and R. Doallo, “Implementing parallel differential evolution on Spark,” in *Applications of Evolutionary Computation. Lecture Notes in Computer Science, Vol. 9598*. Springer, 2016, pp. 75–90.

[28] 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 *Lecture Notes in Computer Science, in press*. Springer, 2016.

[29] A. Gittens, A. Devarakonda, E. Racah, M. Ringenburg, L. Gerhardt, J. Kottalam, J. Liu, K. Maschhoff, S. Canon, J. Chhugani *et al.*, “Matrix factorization at scale: a comparison of scientific data analytics in Spark and C+ MPI using three case studies,” *arXiv preprint arXiv:1607.01335*, 2016.

[30] R. Storn and K. Price, “Differential evolution - a simple and efficient heuristic for global optimization over continuous spaces,” *Journal of Global Optimization*, vol. 11, no. 4, pp. 341–359, 1997.

[31] D. Penas, J. 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.

[32] 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.

[33] C. Moles, P. Mendes, and J. R. Banga, “Parameter estimation in biochemical pathways: A comparison of global optimization methods,” *Genome Research*, vol. 13, no. 11, pp. 2467–2474, 2003.

[34] F. Neri and V. Tirronen, “On memetic differential evolution frameworks: a study of advantages and limitations in hybridization,” in *IEEE Congress on Evolutionary Computation, 2008. CEC 2008.*, 2008, pp. 2135–2142.

[35] Y. Zhai, M. Liu, J. Zhai, X. Ma, and W. Chen, “Cloud versus in-house cluster: evaluating Amazon cluster compute instances for running MPI applications,” in *SC’11: State of the Practice Reports*. ACM, 2011, p. 11.