

 Open access • Posted Content • DOI:10.21203/RS.3.RS-725766/V1

Big Optimization with Genetic Algorithms: Hadoop, Spark and MPI — [Source link](#)

[Carolina Salto](#), [Gabriela F. Minetti](#), [Enrique Alba](#), [Gabriel Luque](#)

Published on: 31 Jul 2021

Topics: [Spark \(mathematics\)](#)

Related papers:

- [Performance Optimization System for Hadoop and Spark Frameworks](#)
- [Choice of Cluster Computing System Hadoop and Apache Spark for Network Systems](#)
- [A Comparative Analysis of Hadoop and Spark Frameworks using Word Count Algorithm](#)
- [An Overview of Hadoop MapReduce, Spark, and Scalable Graph Processing Architecture](#)
- [Using Hadoop Technology to Overcome Big Data Problems by Choosing Proposed Cost-efficient Scheduler Algorithm for Heterogeneous Hadoop System \(BD3\)](#)

Share this paper:    

View more about this paper here: <https://typeset.io/papers/big-optimization-with-genetic-algorithms-hadoop-spark-and-122ey38c1c>

Big Optimization with Genetic Algorithms: Hadoop, Spark and MPI

Carolina Salto

Universidad Nacional de La Pampa

Gabriela Minetti (✉ minettig@ing.unlpam.edu.ar)

Universidad Nacional de La Pampa <https://orcid.org/0000-0003-1076-6766>

Enrique Alba

Universidad de Málaga

Gabriel Luque

Universidad de Málaga: Universidad de Malaga

Research Article

Keywords: Big optimization, Genetic Algorithms, MapReduce, Hadoop, Spark, MPI

Posted Date: July 31st, 2021

DOI: <https://doi.org/10.21203/rs.3.rs-725766/v1>

License: © ⓘ This work is licensed under a Creative Commons Attribution 4.0 International License.

[Read Full License](#)

Big Optimization with Genetic Algorithms: Hadoop, Spark and MPI

Carolina Salto · Gabriela Minetti · Enrique Alba · Gabriel Luque

the date of receipt and acceptance should be inserted later

Abstract Solving problems of high dimensionality (and complexity) usually needs the intense use of technologies, like parallelism, advanced computers and new types of algorithms. MapReduce (MR) is a computing paradigm long time existing in computer science that has been proposed in the last years for dealing with big data applications, though it could also be used for many other tasks. In this article we address big optimization: the solution to large instances of combinatorial optimization problems by using MR as the paradigm to design solvers that allow transparent runs on a varied number of computers that collaborate to find the problem solution. We first investigate the influence of the used MR technology, then including Hadoop, Spark and MPI as the middleware platforms to express genetic algorithms (GAs), arising the MRGA solvers, in a style different from the usual imperative transformational programming. Our objective is to confirm the expected

benefits of these systems, namely file, memory and communication management, over the resulting algorithms. We analyze our MRGA solvers from relevant points of view like scalability, speedup, and communication vs. computation time in big optimization. The results for high dimensional datasets show that the MRGA over Hadoop outperforms the implementations in Spark and MPI frameworks. For the smallest datasets, the execution of MRGA on MPI is always faster than the executions of the remaining MRGAs. Finally, the MRGA over Spark presents the lowest communication times. Numerical and time insights are given in our work, so as to ease future comparisons of new algorithms over these three popular technologies.

Keywords Big optimization · Genetic Algorithms · MapReduce · Hadoop · Spark · MPI

1 Introduction

The challenges that have arisen with the beginning of the era of the Big Data have been largely identified and recognized by the scientific community. These challenges include dealing with very large data sets, since they may well limit the applicability of most of the usual techniques. For instance, evolutionary algorithms, as combinatorial optimization problem solvers, do not scale well to high dimensional instances [20]. To overcome these limitations, evolutionary developers can employ Big Data processing frameworks (like Apache Hadoop, Apache Spark, among others) to process and generate Big Data sets with a parallel and distributed algorithm on clusters and clouds [5, 8, 22, 26, 33]. In this way, the programmer may abstract from the issues of distributed and parallel programming, because the majority of the frameworks manages the load balancing, the

Carolina Salto
Facultad de Ingeniería, Universidad Nacional de La Pampa,
Argentina
CONICET, Argentina
ORCID iD 0000-0002-3417-8603
E-mail: saltoc@ing.unlpam.edu.ar

Gabriela Minetti
Facultad de Ingeniería, Universidad Nacional de La Pampa,
Argentina
ORCID iD 0000-0003-1076-6766
E-mail: minettig@ing.unlpam.edu.ar

Enrique Alba
ITIS Software, Universidad de Málaga, Spain
ORCID iD 0000-0002-5520-8875
E-mail: eat@lcc.uma.es

Gabriel Luque
ITIS Software, Universidad de Málaga, Spain
ORCID iD 0000-0001-7909-1416
E-mail: gabriel@lcc.uma.es

network performance, and the fault tolerance. These features made them popular, creating a new branch of parallel studies where the focus is on the application and not on exploiting the underlying hardware.

A well-known computing paradigm that is used to process Big Data is MapReduce (MR). It splits the large data set into smaller chunks in which the *map* function processes in parallel and produces key/value pairs as output. The output of *map* tasks is the input for *reduce* functions in such a way that all key/value pairs with the same key go to the same *reduce* task [5]. Hadoop is a very popular framework, relying in the MR paradigm [1,34], both in industry and academia. This framework provides a ready-to-use distributed infrastructure, which is easy to program, scalable, reliable, and fault-tolerant [14]. Since Hadoop allows parallelism of data and control, we research for other software tools doing similar jobs. The MapReduce-MPI (MR-MPI) [24] is a library built on top of MPI, which conforms another framework with a somewhat similar goal. Here you can have more control of the platform, allowing to improve the bandwidth performance and reduce the latency costs. Another popular Big Data framework is Apache Spark [13] that is different from Hadoop and MR-MPI, since the computational model of Spark is based on memory. The core concept in Spark is Resilient Distributed Dataset (RDD) [14], which provides a general purpose efficient abstraction for distributed shared memory. Spark allows developing multi-step data pipelines using a directed acyclic graph.

Although the three mentioned technologies allow implementations following the MR paradigm, they have significant differences. Consequently, they encourage us to carry out a performance analysis targeted to discover how big optimization can be best implemented onto the MR model that later is run by any of these three platforms. This comparative analysis arouses interest for any curious scientist, in order to offer evidence about their relative performance (advantages and disadvantages). Moreover, the MR paradigm can contribute to build new optimization and machine learning models, in particular scalable genetic algorithms (MR-GAs), as combinatorial optimization problem solvers, which are widely used in the scientific and industrial community. In the literature, many researchers have reported on GAs programmed on Hadoop [5,8,11,32,33] and Spark [15,22,26], and a few ones under MR-MPI [29], according to the authors knowledge. Moreover, these proposals present different GA parallel models for big optimization, but they are specific for a particular MR framework. Furthermore, these research works mainly focus on the parallelization of highly time-consuming fitness computation, but not on solving prob-

lems whose complexity is associated with handling Big Data. All this implies a significant lack of information on the advantages and limitations of each framework to implement MRGA solvers for big optimization. In this sense, the selection of the most appropriate one to implement this kind of algorithm results in a very complex task. In order to mitigate the lack of information about the MRGA scalability on the three most known MR frameworks (MR-MPI, Hadoop, and Spark), we define the following research questions:

- *RQ1: Can we efficiently design big optimization MRGA solvers using these frameworks?*
- *RQ2: Which of the frameworks allows the MRGA solver to reach its best time performance by scaling to high dimensional instances?*
- *RQ3: Are MRGAs scalable when considering an increased number of the map tasks?*
- *RQ4: Is the time spent in communication a factor to consider when choosing a solver?*

With the first research question, we analyze the usability of these frameworks to design MRGAs that solve big optimization problems. The RQ2 deepens this analysis, hopefully offering interesting information on the MRGA performance when the instance dimension scales. Furthermore, the scalability of all the studied approaches is also analyzed considering the number of parallel process (map tasks), as RQ3 suggests. Finally, the last research question allows us to examine which MRGA solver spends more time in communication than in computation.

To address these *RQs*, we analyze how a Simple Genetic Algorithm (SGA) [12] can take advantage of these Big Data processing frameworks in the optimization of large instances of a problem. We here decide to use this SGA because because it is a canonical technique in the core of the Evolutionary Algorithm (EAs) family, and most things done on it can be reproduced in other EAs and population-based metaheuristics. For the purposes of this analysis, in this research a SGA design is tailored for the MR paradigm, procuring the so called MRGA [29], coming out from a parallelization of each iteration of a SGA under the Iteration-Level Parallel Model [31]. The contributions of this work are manifold. We develop the same optimizer (MRGA) using three open-source MR frameworks. We consider the implementations made in our previous research [29], MRGA-H for Hadoop and (MRGA-M) for MR-MPI. Moreover, in this work, the MRGA design is implemented into the Spark framework, arising the MRGA-S algorithm. Later on, we analyze and compare these three implementations considering relevant aspects such as execution time, scalability, and speedup to solve a large

1 problem size of industrial interest as the knapsack prob-
2 lem [10]. As to our knowledge, this is the first work
3 considering the same MRGA solver implemented in the
4 three widely known platforms and pointing out their
5 different features for the benefit of future researches.
6

7 This article is organized as follows: next section dis-
8 cusses the MR paradigm and Big Data frameworks,
9 showing their similarities and differences. Section 3 pre-
10 sents a brief state of the art in implementing GAs with
11 the MR paradigm, and contains our proposal. Sections
12 4 and 5 define meaningful experiments to reveal infor-
13 mation on the three systems, perform them, and give
14 some findings. Finally, Section 6 summarizes our con-
15 clusions and expected future work.
16

17 2 MR Paradigm and Frameworks

21 An application in the MR paradigm is arranged as a
22 pair (or a sequence of pairs) of *map* and *reduce* func-
23 tions [7]. Each *map* function takes as input a set of
24 key/value pairs (records) from data files and generates
25 a set of intermediate key/value pairs. Then, MR groups
26 together all these intermediate values associated with a
27 same intermediate key. A value group and its associated
28 key is the input to the *reduce* function, which combines
29 these values in order to produce a new and possibly
30 smaller set of key/value pairs that are saved in data
31 files. Furthermore, this function receives the intermedi-
32 ate values via an iterator, allowing the model to handle
33 lists of values that are too large to fit into main memory.
34 The input data is automatically partitioned into a set
35 of M splits when the *map* invocations are distributed
36 across multiple machines, where each input split is pro-
37 cessed by a *map* invocation. The intermediate key space
38 is divided into R pieces, which are distributed into R
39 *reduce* invocations. The number of partitions (R) and
40 the partitioning function are user defined.
41

42 As previously mentioned, our aim in this work is to
43 perform a comparison of the different Big Data frame-
44 works to develop big optimization MRGA solvers. For
45 that purpose, this section presents three Big Data frame-
46 works, as the Hadoop [1], the MR-MPI [24], and Spark
47 [13], with the goal of identifying the advantages and
48 limitations of each one. In this process, the focus is on
49 the installation, use, and productivity characteristics of
50 each framework.
51

52 2.1 Hadoop

53 The Hadoop framework consists of a single master **Re-**
54 **sourceManager**, one slave **NodeManager** per cluster-node,
55
56
57

and a **MRAppMaster** per application, which is imple-
mented using the Hadoop YARN framework [1]. In or-
der to meet those goals, the central Scheduler (in the
ResourceManager) responds to a resource request by
granting a container. Essentially, the container is the re-
source allocation, which allows to an application the use
of a specific amount of resources (memory, CPU, etc.)
on a specific host. In this context, the Hadoop client
submits the job/configuration to the **ResourceManager**
that distributes the software/configuration to the slaves,
schedules, and monitors the tasks, providing status and
diagnostic information to the client including fault tol-
erance management. In this sense, it is noticeable that
the installation and the configuration of the Hadoop
framework requires a very specific and long sequence of
steps, becoming difficult to adapt it to a particular clus-
ter of machines. Moreover, at least one node (master)
is dedicated to the system management.

To deal with parallel processing applications on large
data sets, Hadoop incorporates the Hadoop Distributed
File System (HDFS) and Hadoop YARN. The first one
handles scalability and redundancy of data across nodes.
The second one is a framework for job scheduling that
executes data processing tasks on all nodes.

58 2.2 MR-MPI

MR-MPI is a small and portable C++ library that only
uses MPI for inter-processor communication, thus the
user writes a main program that runs on each proces-
sor of a cluster, making *map* and *reduce* calls to the
MR-MPI library. As a consequence, a new framework
arises with no extra installation and configuration tasks
(light management and easy to program with it), but
not fault tolerance. The use of the MR library within
MPI follows the traditional mode to call the *MPI_Send*
and *MPI_Recv* primitives between pairs of processors,
using large aggregated messages to improve the band-
width performance and reduce the latency costs.

59 2.3 Spark

Apache Spark has a very powerful and high-level API,
which is built upon the basic abstraction concept of
the Resilient Distributed Dataset [35]. A RDD is an
immutable and a fault-tolerant collection of elements
in shared memory that can be operated on parallel.
This kind of datasets is divided into logical partitions,
each one is computed on different nodes of the cluster
through operations that transform a RDD (creating a
new one) or perform computations on the RDD (re-
turning a value).

Spark applications are composed of a single **driver** program and multiple workers or **executors**. The client process starts the **driver** program, which orchestrates and monitors execution of a Spark application and calls to actions. With each action, the Spark scheduler builds an execution graph and launches a Spark job. Each job consists of stages, which are a collection of tasks that represent each parallel computation and are performed on the executors (Java Virtual Machine, JVM, processes). Each **executor** has several task slots for running tasks in parallel. The physical placement of **executor** and **driver** processes depends on the cluster type and its configuration.

In order to run on a cluster, the `SparkContext` can connect to several types of cluster managers (either Sparks own standalone cluster manager, Mesos or YARN), which allocate resources across applications. In this work, the Hadoop YARN is adopted as cluster manager, due to their previous use with Hadoop. The installation and the configuration of the Spark framework is not direct, requiring the configuration of many properties that control internal settings. Most of them have reasonable default values, but others require to be adjusted to an appropriate values and generally are particular to the cluster features. These parameters vary from Spark's properties to size's settings of the JVM.

2.4 Final Discussion on the Platforms

Spark allows a more flexible organization of the processes, appropriate for iterative algorithms, and eases efficiency due to the use of in memory data structures (RDDs). But Spark requires a quite large amount of RAM memory and it is quite greedy in the utilization of the cluster resources, while Hadoop and MR-MPI can be used in low-resource and non-dedicated platforms. Hadoop is designed mainly for batch processing, while with enough RAM, Spark may be used for near real-time processing. Also, many problems in industrial domains are implemented in C/C++, which are only natively supported by Hadoop and MR-MPI implementations (and not in Spark). Therefore, the research on efficient uses of these first frameworks (as done in this article) is today an important domain [5,8].

Hadoop framework stores both the input and the output of the job in the HDFS, whereas MR-MPI allocates pages of memory. Spark can also use HDFS to store the data, providing fault tolerance by the task duplication. In this way, Hadoop and Spark also supply data redundancy. But the HDFS creation and its configuration require a careful setting of properties, resulting in a time-consuming process. Instead, MR-MPI is not able to detect a dead processor and retrieve the

data, being the MPI implementation responsible for detecting and handling network faults.

Spark uses lazy evaluation to reduce the number of passes it has to take over our data by grouping operations together. In platforms like MR-MPI and Hadoop, developers often have to spend a lot of time considering how to group together operations to minimize the number of MR passes. In Spark, there is no substantial benefit to writing a single complex map instead of chaining together many simple operations. Thus, users are free to organize their program into smaller, more manageable operations.

Table 1 shows a comparison between the considered frameworks taking into account various aspects: such as language supported, volume of data sets, processing type, easy configuration, among others.



3 Big Optimization with Genetic Algorithms

In this section, we start with a review of the literature about how GAs were translated into different Big Data frameworks. After that, we describe the simple model of SGA used in this work and how it is adapted to be implemented by following MapReduce (MRGA). The idea is to implement the same MRGA using the Hadoop (MRGA-H), MR-MPI (MRGA-M) and Spark (MRGA-S) frameworks to solve big optimization problems. This will allow us to compare the results and to find out their strong and weak features. The implementations of MRGA-H and MRGA-M algorithms are obtained from [32] and own previous work [29], respectively. In the case of MRGA-S, its implementation was developed from scratch.

3.1 Literature Review

Some of the most representative works that model GAs using Big Data frameworks are described in this section. Verma et. al [32,33] proposed a SGA and a CGA based on the selecto-recombinative GA, proposed by [12], which only use two genetic operators: selection and recombination. SGA was developed using the Hadoop framework. The authors match the *map* function with the evaluation of the population fitness, whereas the *reduce* function performs the selection and recombination operations. They proposed the use of a custom-made Partitioner function, which splits the intermediate key/value pairs among the reducers by using a random shuffle. In [11], the authors proposed a similar model of GA than Verma et. al [32] for software testing. The main difference relays in the use of only one reducer that receives the entire population. Thus,

Table 1 Comparison between Big Data frameworks.

Features		MR-MPI	
open-source	yes	yes	yes
popular for big-data	yes	no	yes
language support	Java, C++, Ruby, Python	C, C++, Python	scala, Java, Python, R
fault-tolerance	yes	no	yes
processing approach	read and write to disk	read and write to disk	in-memory
volume of data sets	huge	large (RAM + disk)	quite large (memory sizes)
processing type	batch	batch	near real-time
iterative processing	no	yes	yes
load balancing	automatic	manual	automatic
installation	easy	easy	easy
configuration	relatively difficult	easy	relatively difficult

the reducer can perform the selection and apply the crossover and mutation operators to produce a new offspring to be evaluated in the next MR job. However, different parallel GA's models were proposed by Ferrucci et al. [8] using Hadoop as distributed infrastructure. The authors propose a global model, a grid model, and finally an island model. They analyze the proposals in terms of execution time and speedup, as well as the behavior of the three parallel models in relation to the overhead produced using Hadoop. Chavez et al. [6] introduce changes in ECJ [30] to follow the MP paradigm in order to launch any EA problem on a big data infrastructure using Hadoop similarly as when a single computer is used to run the algorithm. Jatoth et al. [16] solved the problem of QoS-aware big service composition by implementing a MapReduce based evolutionary algorithm with guided mutation on a Hadoop cluster, which use a global model in the MapReduce phase.

An implementation of GAs using Spark can be found in [22]. Their proposal consists in the partition of the population in many worker processes which applies the genetic operations and evaluation by the use of *map* function, when the stop criterion is met, a *reduce* function aggregates the subpopulations to find the most promising individuals. In the same line of using Spark as parallel platform, Hu et al. [15] use a SGA as optimizer tool, where the population is divided in chunks for evaluation purposes by using a *map* function. After that, a *collect* function is used to gather all the individuals of the population together to apply genetic operations. More recently, two versions of parallel GAs were proposed in [4] using Spark framework. The proposals are based on the traditional master slave model and the island model to solve large dimensional classifier problems. The first model handles the evolutionary process by the Spark *driver*, which sends the individuals across the executors to compute the fitness. In the second one, each island is an *executor* and evolves a subpopula-

tion. The proposed models are evaluated in relation to performance and accuracy over multiple cluster sizes.

Many of the reviewed works deploy computing-intensive runs of EAs on the Big Data infrastructures [8,6,16]. In particular, they implement parallel versions of EAs to optimize the running time for the algorithmic experiments, because the optimization problems have computationally costly fitness evaluation functions. As to papers dealing with large data volume, we can mention the work of Verma et al. [32,33], which involves 10^n ($n = 4$) variables and a population of $n \times \log n$ size. Finally, Chavez et al. [6] and Alterkawi et al. [4] address large and complex data classification tasks, but the authors do not indicate the amount of memory usage.

The aforementioned proposals present different GA parallel models for big optimization, but they are specific for a single MR framework. This implies a significant lack of information on the advantages and limitations of each framework to implement GAs for big optimization. In this sense, the selection of the most appropriate one to implement this kind of algorithm results in a very complex task. In order to mitigate this lack of information, the main objective of our research is to design and implement a scalable GA on the three most known MR frameworks: MR-MPI, Hadoop, and Spark.

3.2 Big Optimization MRGA Solver

For our study, we will use a simple genetic algorithm, SGA, whose operations can be found in a great number of Evolutionary Algorithms in the literature. Our aim is then to guide future research that is linked to these search operations when designing other algorithms to MR implementations. The pseudocode of SGA is presented in the Algorithm 1, which starts by generating an initial population. During the evolutionary cycle, the population is evaluated and then a set of parents is selected by tournament selection [21]. After that, the uni-

Algorithm 1 Sequential GA

```

1:  $t = 0$ ; {current generation}
2: initialize( $Pop(t)$ );
3: evaluate( $Pop(t)$ );
4: while (non stop criterion is met) do
5:    $Pop'(t) = \text{select}(Pop(t))$ ; { $k$ -wise tournament selection without replacement}
6:    $offspring = \text{recombine}(Pop'(t), p_c)$ ; {uniform crossover}
7:    $Pop(t+1) = \text{replace}(Pop(t), offspring)$ ;
8:   evaluate( $Pop(t+1)$ );
9:    $t = t + 1$ 
10: end while
11: return (best individual);

```

form recombination operator is applied to them. The recently created offspring conform the new population for the next generation (using the generational replacement). The evolutionary process ends when either the optimum solution to the problem at hand is found or the maximum number of iterations is reached.

Our proposed MRGA algorithm preserves the SGA behavior, but it resorts to parallelization for some parts: the evaluation and the application of genetic operators. Although, our technique performs several operations in parallel, its behavior is equal to the sequential GA.

A key/value pair has been used to represent individuals in the population as a sequence of bits. To distinguish identical individuals (with the same genetic configuration), a random identifier (ID) is assigned in the *map* function to each one. The ID prevents that identical individuals were assigned to the same *reduce* function, in the phase of shuffling when the intermediate key/multivalued space are generated. The sequence of bits together with the ID corresponds to the key in the key/value pair. The value part is the individual fitness, which is computed by the *map* function.

For large problem sizes, the population initialization could be a consuming time process. The situation can get worse with large individual sizes, as the case in this work. According to this situation, this initialization is parallelized in a separate MR phase. The *map* functions are only used to generate random individuals. After that, the iterative evolutionary process begins, where each iteration consists of a *map* and *reduce* functions. The *map* functions compute the fitness of individuals. As each *map* has assigned different chunk of data, they evaluate a set of different individuals in parallel. This fitness is added as value in the key/value pair. Each *map* finds their best individual that is used in the main process to determine if the stop criterion is met. The *reduce* functions carry out the genetic operations. The binary tournament selection is performed locally with the intermediate key space, which is distributed in the partitioning stage after *map* operation. The uniform crossover (UX) operator is applied over the selected individuals. The generational replacement

is implemented to build the new population for the next MR task (a new iteration).

Regardless of the framework used, the key/value pairs, generated at the end of the *map* phase, are shuffled and split to the *reducers* and converted in an intermediate key/multivalued space. The shuffle of individuals consists in a random assignment of individuals to reducers instead of using a traditional hash function over the key. This modification, as suggested in [32], responds to avoid that all values corresponding to a same key (identical individuals) will be sent to the same *reduce* function, generating a biased partition and fix assigned of individuals to the same partition through evolution and an unbalance load of *reduce* functions at the end of evolution. Therefore, the intermediate key/value pairs are distributed into R partitions using an uniform distribution.

3.3 MRGA-H Algorithm

Some modifications were introduced in the code developed in [32]. The most important ones are related to the changes imposed by passing from the old MRV1 to the new Java MapReduce API MRV2 [1], because they are not compatible with each other. These important differences involve the new package name, the context objects that allow the user code to communicate with the MR system, the Job control that is performed through the Job class in the new API (instead of the old one *JobClient*), and the *reduce()* method that now passes values as an *Iterable Object*. Also, some modification were required during the generation of the random individual ID. Finally, the individual evaluation was included in the method fitness of the *GAMapper Class*. The rest of the code with the functionality of the GA remains without important modifications. The scheme of MRGA-H is plotted in Figure 1. Chunks of data read from HDFS and processed by each map are represented by shaded rectangles.

3.4 MRGA-M Algorithm

MRGA-M creates the MPI environment for the parallel execution. Then, the sequence begins with the instantiation of an MR object and the setting of their parameters. The MRGA-M follows the scheme shown in Figure 2 where boxes with solid outlines are files and the chunks of data processed by each map are represented by shaded rectangles in a hard disk.

The first MR phase consists of only one *map* function (calling to a serial *Initialize()*) to create the initial population. In our implementation the main process

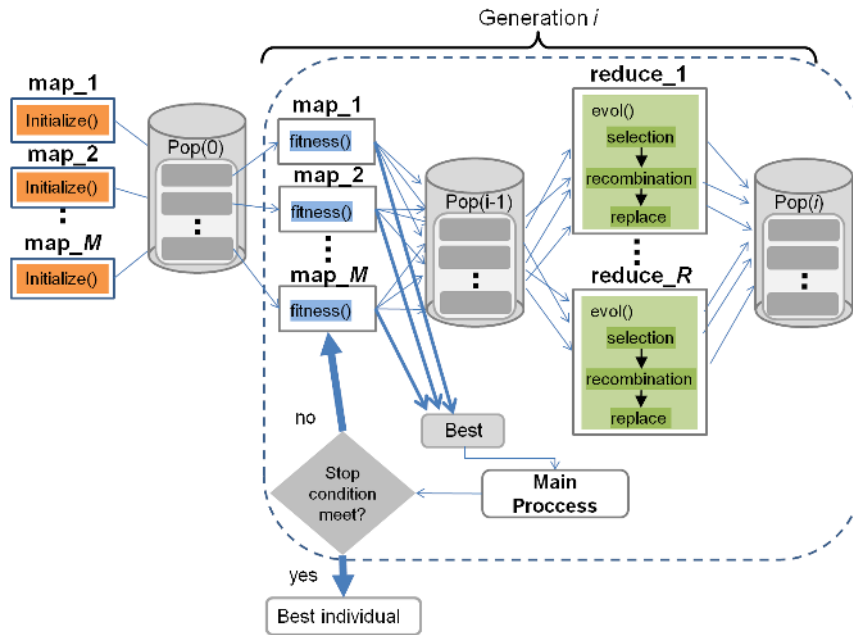


Figure 1 MRGA-H scheme.

(process with MPI id equal to zero) generates a list of filenames. Our *Initialize()* function processes each file to build the initial population.

The second and following MR phases have a sequence of *map* and *reduce* functions. These *map* functions receive a chunk of the large file passed back to our *fitness()* and then split it in M chunks. The *fitness()* function processes each key (an individual) received, evaluates it obtaining the fitness value and emits a key/value pairs. After that, the MR-MPI *aggregate()* function shuffles the key/value pairs across processors by distributing the pairs randomly. Then, the MR-MPI *convert()* function transforms a key/value pairs into a key/multi-value pairs. Finally, the *Evol()* function (from the *reduce* method) will be called once for each key/multi-value pair assigned to a processor. This function selects a pair of individuals by tournament selection and performs the recombination. The new individuals generated are written into permanent storage to be read by the *map* methods in the following MR phase.

3.5 MRGA-S Algorithm

As we have before explained, Spark extends and generalizes the MR idea with a different implementation. In consequence, we need to introduce changes to the MRGA design to obtain the MRGA-S, which are detailed in the following.

The proposed MRGA is based on Spark RDDs to store the population. This RDD is cached in memory to

accelerate the processing instead of using files to store the population, as in MRGA-H and MRGA-M. However, the MRGA-S also exploits the parallelism in the evaluation and in the application of genetic operators. Consequently, MRGA-S follows the same logical functionalities than both MRGA-H and MRGA-M, with respect to the SGA behavior.

In this MRGA implementation, a different conception of key/value pairs to represent individuals is used. The key represents the partition were an individual has to be assigned to, whereas the value correspond to the individual itself.

Figure 3 presents a scheme of the MRGA-S. The sequence begins with the creation of a RDD (Stage 1), which is parallelized in the main program. The elements of the RDD are copied to form a distributed dataset that can be operated in parallel in each worker, which transforms them and returns the results to the main program. After that, an iterative process begins consisting of two Spark stages that are repeated until the stop criterion is met.

The first step in the main loop (Stage k) assigns a random value to each individual in the range $[1, \dots, R]$, by using a special version of the *map* operation (*mapToPair()* operation). This conversion prepares a RDD for the next operation, which consists in grouping the individuals with the same key. Note that this step (Stage k) is equivalent to the **Partitioner** in the MRGA-H or to the *aggregate()* function in MRGA-M.

The next Stage $k+1$ begins with the redistribution of the individuals across the partitions, by using the

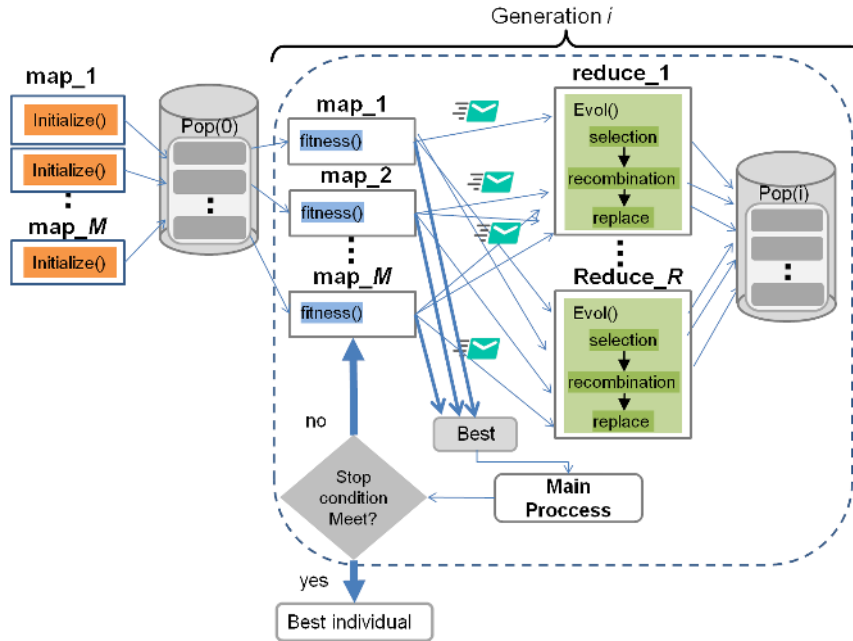


Figure 2 MRGA-M scheme.

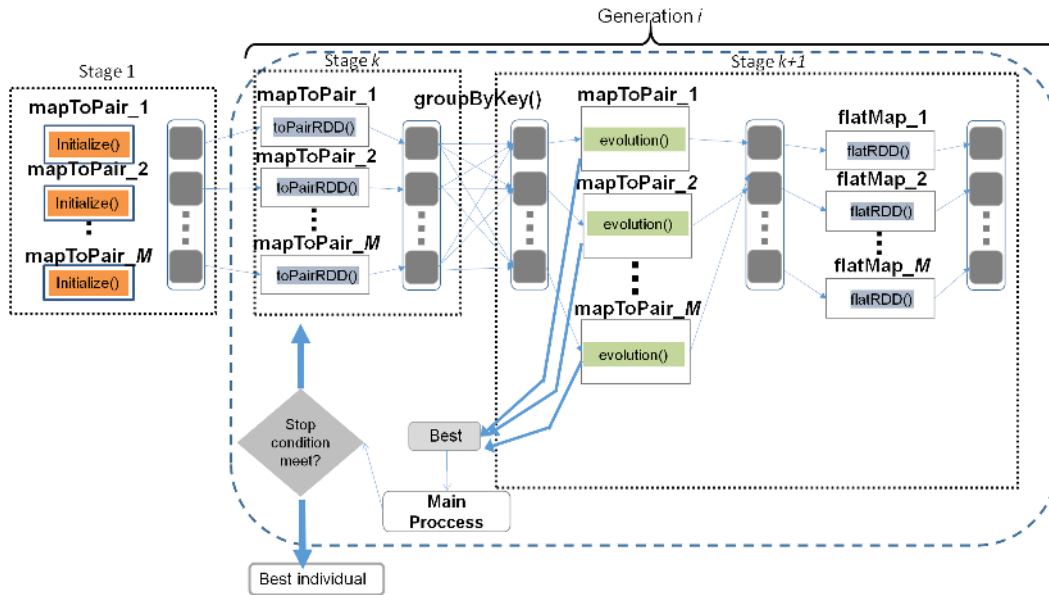


Figure 3 MRGA-S scheme.

groupByKey() function. After that, a *mapToPair()* operation is invoked with the *Evolution()* function as its parameter, in order to evaluate the individuals and apply genetic operators into a partition, generating the new individuals for the next generation. Although, this is a new difference with MRGA-H and MRGA-M, MRGA-S maintains the SGA's underlying idea. Finally, a new RDD containing the individuals from all the partitions (the whole population) is obtained to continue with the first step of Stage $k+1$ in this iterative process.

4 Experimental Setting

To address the research questions about the efficiency and scalability presented in Section 1, we consider as a benchmark the knapsack problem [18,25,28,36] to evaluate the proposed algorithms. The choice of this problem was motivated by the fact that it allows us to assess the MRGA scalability on different big instance sizes. To carry out the evaluation of the analysis of our proposals, we use metrics such as execution time, scalability, speedup, and communication vs. computation.

The problem, the experimentation methodology, and the evaluation metrics are explained in the following subsections.

4.1 Knapsack Problem

The knapsack problem (KP) is a classic NP-complete problem [18], which is defined by the task of taking a set of items, each with a weight and a profit, filling the knapsack so that the total profit is maximized, but not exceeding the maximum weight the knapsack can hold. The KP formulation is shown in Equation 1.

$$\max \sum_{i=1}^{N_s} x_i p_i \quad (1)$$

subject to

$$\sum_{i=1}^{N_s} x_i w_i \leq K$$

where K is the maximum weight the knapsack can hold, and N_s is the number of items in the set, S . Each item has a weight w_i and a profit p_i . Here x_i indicates whether an item i is present or not in the knapsack. Therefore, a KP solution is represented by a bit string in MRGA, as is shown in Figure 4 and its implementation is described in Section 3.2.

KP is a very well-known problem in computer science. It occurs in many situations be they in industry, communication, finance, applied sciences or in real life [9,17,19,27], being itself a very interesting combinatorial problem to be dealt using the big optimization solver presented in this work. In general, the KP literature solves problem instances that vary between 100 and 10,000 items in the knapsack. In this article, we propose to optimize six different high dimensional instances with a very large number of items: 25,000, 50,000, 75,000, 100,000, 125,000, 150,000, 200,000 and 300,000 items. They are named as 25K, 50K, 75K, 100K, 125K, 150K, 200K and 300K, respectively. These big KP instances were obtained with the generator described in [23] and can be found in the repository <https://github.com/GabJL/LargeKPIInstances>, choosing the uncorrelated data instances type (no correlation between the weight and the profit of an item). We selected these large datasets because they represent different degrees of computational and memory load, being also an important contribution to the state-of-the-art of the problem of the knapsack.

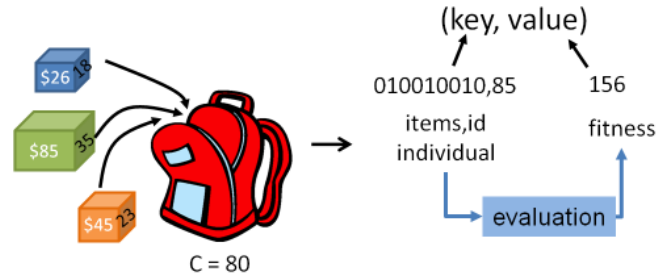


Figure 4 Solution representation of the knapsack problem in MRGA.

4.2 Experimentation Methodology

Each MRGA's approach evolves 50000 randomly initialized individuals. This population size was chosen in order to increase the memory load that our big optimization solver has to manipulate. For each generation, these algorithms use the binary tournament selection to select parents, a probability of 100% to recombine the parents using the UX operator, and the generational replacement to obtain the next population. Let us recall that for each considered KP instance and number of *map* tasks (*#map*), we execute 30 times the MRGA-M, MRGA-H, and MRGA-S. The computational environment used in this work to carry out the experimentation is a cluster of five nodes with 8 GB RAM.

The sequence of bits of an individual is grouped by arrays of long ints (64 bits) and their lengths depends on the instance dimension. For example, the individual length for the 25K instance is 392 long ints (25,000/64) requiring 3.1 KB of memory, and therefore the population demands 156.25 MB. The decision of using long ints was to optimize the bit operations required by the evolutionary operators. In this way, the total RAM requirements varies from 150 MB to 1.8 GB, justifying the use of Big Data frameworks.

4.3 Evaluation Metrics

In the previous section we explained the methodology for experimentation, now we will develop on the strategy to carry out a fair comparison between the MRGA solvers. In this way, distinct metrics are considered to evaluate them, such as execution time, scalability, and speedup. This becomes as a good practice to report results in the metaheuristic field [3]. We also evaluate the behavior of our proposals considering different number of maps and reducers in the case of MRGA-H and MRGA-M and workers for MRGA-S, in order to assess an analysis of the implications of the amount of parallelism in the performance of MRGA approaches.

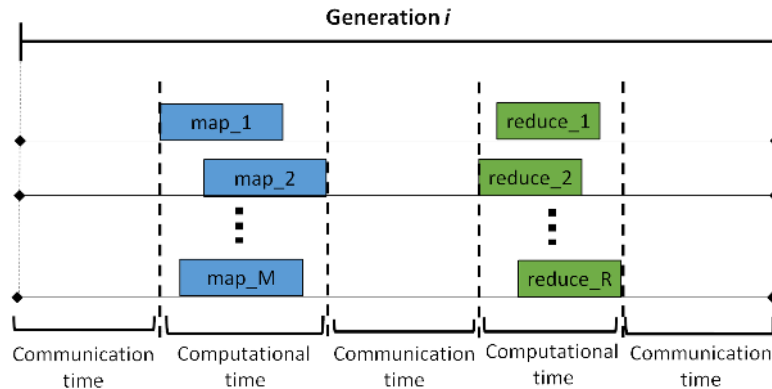


Figure 5 The method to compute times for MRGA-M and MRGA-H.

Execution Time. The execution time (or runtime) achieved by each MRGA approach is measured in milliseconds using the system clock. This includes the time between starting and finishing the whole algorithm. Consequently, we include all the communication involved in the execution. As a way to analyze the implications of the amount of parallelism in the execution of each approach, a comparison among the different MRGA solvers is carried out. This is the base metric to measure the scalability and speedup that are explained below.

Scalability. We analyze the scalability from two different dimensions. The first one refers to the algorithm capacity to solve increasing sizes of the problem. For that reason we include six different instances in the study. In this case, we maintain the number of *map* tasks constant. The second one addresses to increase the number of *map* tasks whereas we keep the problem size fixed, considering 4, 8, and 12 *map* tasks. This study allows us to determine how the execution times are modified when more resources to solve the same problem are available. Consequently, we have scalability with an increasing problem size and scalability with a constant overall load.

Speedup. The speedup (s_m) is the ratio between the mean execution time on one processor and the mean execution time on m processors. We use the definition of weak speedup given in [2] that compares the execution time of the parallel algorithm on one processor against the execution time of the same algorithm on m processors. For this particular study, the solution quality is taken as the stopping criterion. The evaluated MRGA solvers should compute solutions having a similar accuracy. Thus, a relaxation of the optimal fitness value for each KP instance (e.g., 90%) is considered, but in any case the same value. All these define an orthodox speedup measure in the Alba's taxonomy [2].

Communication vs. Computation. A study about the communication and computation times of each algo-

rithm allows us to understand the reasons that causes our proposals have a slightly improved speedup. We adopt the method used in [8], which is proposed for the Hadoop framework and we have extended for the other two ones. Figure 5 illustrates how the communication and computational times are calculated per generation. This method allows to isolate the GA execution time (computational time) from the time spent by each framework to put on-line and run each algorithm (communication time).

5 Result Analysis

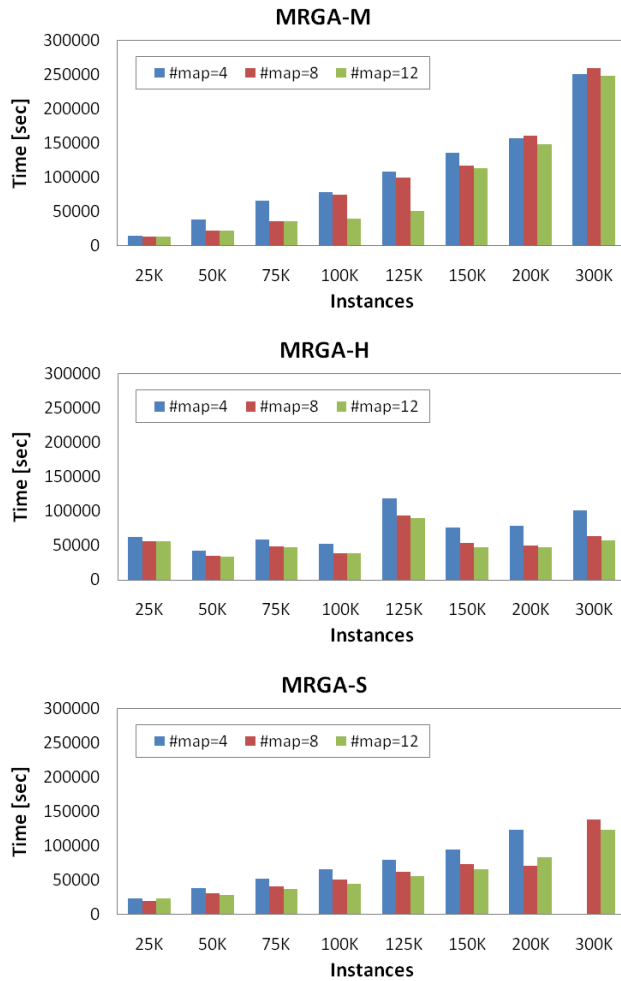
In this section, we present the results that allows as to answer the different *RQs* formulated in Section 1. The comparison between the MRGA solvers with respect to the scalability is in Section 5.1. The analysis of the speedup is in Subsection 5.2. Finally, in Subsection 5.3, we contrast the communication and computation times consumed by each MRGA.

5.1 Scalability

Table 2 and Figure 6 show the execution times achieved for each algorithm by increasing the problem dimension. In Table 2, the MRGA-S execution time for the 300K instance is not available (N/A) because this solver was not able of running such a large instance in our systems. For every instance, each bar of Figure 6 represents a different number of *map* tasks. As expected, the execution times increase as the dimensionality of the problem grows. This situation is very clear in the case of MRGA-M and MRGA-S. However, MRGA-H presents a behavior with no direct dependence of the problem size. The previous results give us support to answer the *RQ2* since these algorithms can solve efficiently incremental high dimensional instances, becoming scalable

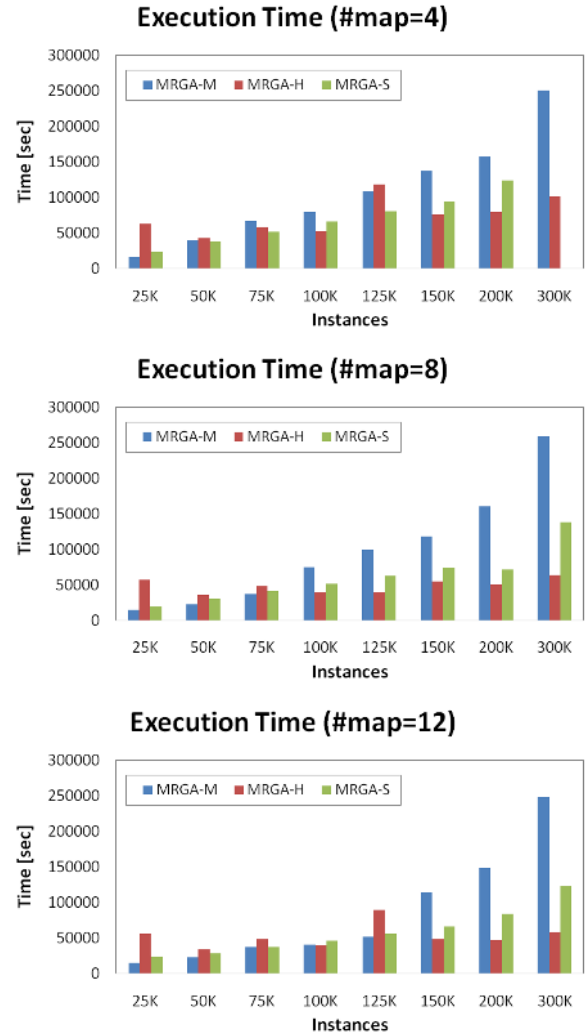
Table 2 Execution times achieved for each MRGA solver.

Inst	MRGA-M			MRGA-H			MRGA-S		
	#map=4	#map=8	#map=12	#map=4	#map=8	#map=12	#map=4	#map=8	#map=12
25K	15420	14025	14130	62862	56797	55978	22951	19391	23349
50K	39039	22424	22710	42607	35255	33931	38107	30232	28032
75K	66585	36600	36570	58439	48221	47597	51448	40955	37335
100K	78894	74724	40150	52234	38541	38479	65415	50923	45181
125K	108645	99390	51810	118170	94092	89665	79798	61931	55903
150K	136815	118110	113798	75931	54267	48066	94052	73351	66071
200K	157475	160672	148570	79271	50402	47387	123655	71088	83443
300K	251028	260218	248944	101026	63380	57806	N/A	172853	208043

**Figure 6** Mean execution time of MRGA algorithms to solve the KP instances.

big optimization solvers. In particular, the MRGA-H presents the best performance.

Now, if we analyzed what happens when a same KP instance is solved by some MRGA and the number of *map* tasks is increased, as shown in Figure 6, we can observe a decrease in the execution time. This study allows to infer how is affected the MRGA execution times when the same load is maintained but the amount of *map* tasks is augmented. This suggests that when more resources are used to solve the same problem, we

**Figure 7** Mean execution time of MRGA algorithms for each number of map tasks to solve the KP instances.

obtain a gain in the time. Being, 12 a good number of *map* tasks for MRGA-M, while 8 *map* tasks is enough for obtaining accurate results in both MRGA-H and MRGA-S and the improvement achieved adding more maps is meaningless.

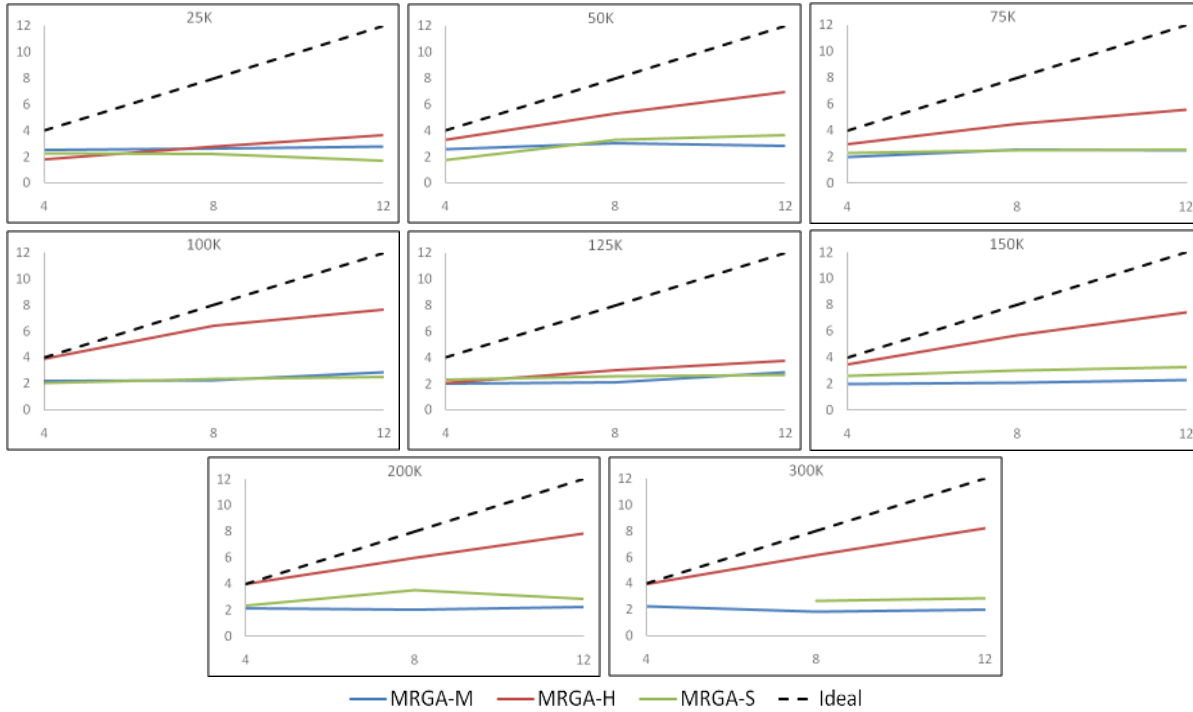


Figure 8 Speedup trend per instance.

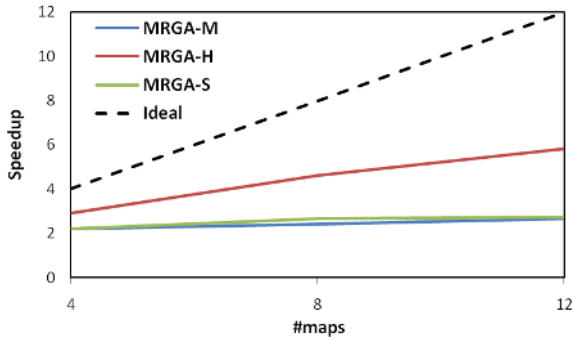


Figure 9 Mean speedup per MRGA solver.

5.2 Speedup

We analyze the results shown in Figure 7 by comparing the execution times of each MRGA. In this way, we find responses to the research questions *RQ1* and *RQ3* about the MRGA efficiency and scalability when more computational resources to solve the same problem are available. For the smallest datasets, i.e. instances with less than 100,000 items, we observe that the execution of MRGA-M is always faster than the executions of the remaining MRGAs, regardless of the number of *map* tasks used. However, for the largest data sets, MRGA-M is the slowest MRGA solver, mainly when 4 and 8 *map* tasks are employed, while MRGA-H is the fastest one. The MRGA-M weak behavior is caused by the

hardware resource limitations to support big instances in a few number of *map* tasks. Moreover, MRGA-H is the algorithm with less time variations than the other two MRGAs for a given *map* task number. In the case of MRGA-S, we observe a slight increasing in the runtimes when instances with more number of items are solved. Consequently, we can infer that the three MRGAs present an efficient performance and are scalable, being MRGA-H the most efficient and scalable solver for big optimization. This conclusion can also be deduced from Figure 6, although it cannot be seen with the naked eye.

Now, we focus on the speedup values to reinforce the justification of the previous answer to *RQ1* from a different point of view. Figure 8 graphically shows the speedup values for each KP instance. The MRGA-H speedup is the best of the three algorithms for the majority of the problem instances. Although the speedup is sub-linear ($s_m < m$), the MRGA-H results are quite good because they are approximately at 0.65 from the ideal speedup value for 4 and 8 *map* tasks. Both MRGA-M and MRGA-S present a poor relation respect to the ideal value, and this situation becomes worse when larger number of *map* tasks is considered (the values are very small, less than 0.3). These observations are corroborated with the average speedup values per MRGA solvers that is shown in Figure 9.

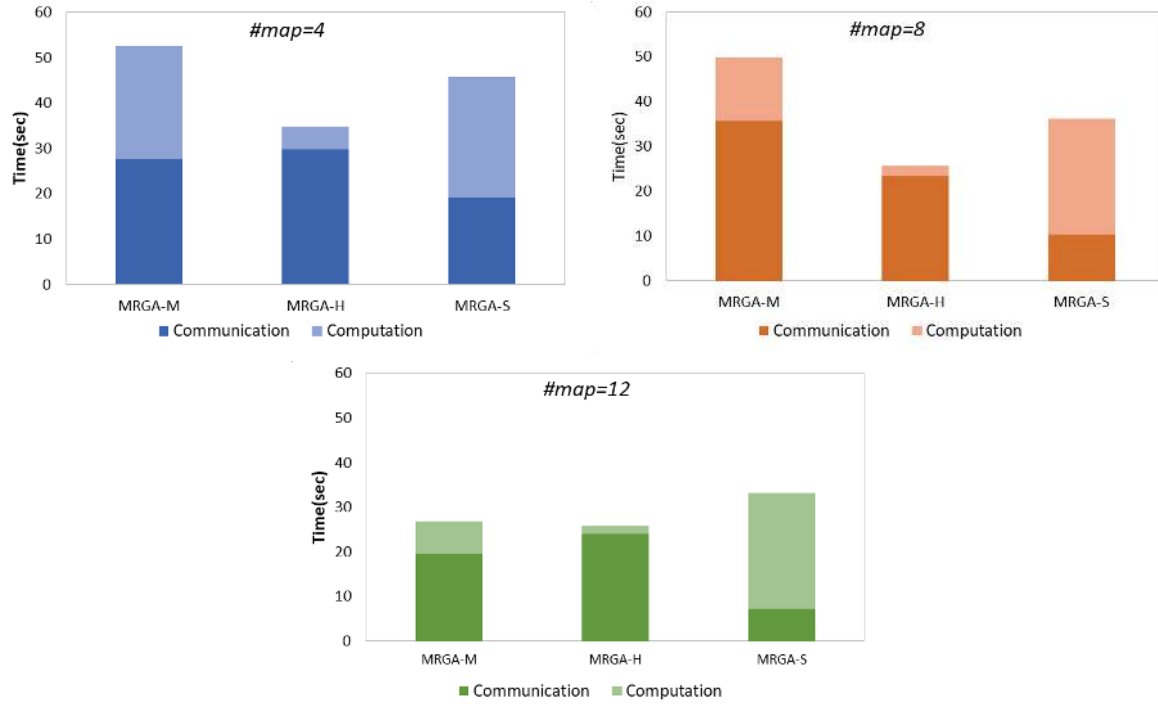


Figure 10 The average communication vs. computational times spent by each algorithm per generation.

5.3 Communication vs. Computation

Figure 10 shows the communication and computation times for each MRGA solver taken as reference the instance with a dimension of 100,000 items. Similar observations can be done for the other instances. This kind of analysis allows us to give more details about the execution time achieved by each approach. The stacked bars represent communication and computational times for a generation.

On the one hand, it is worth noting that MRGA-S presents the lowest communication and it goes decreasing as the number of maps is increased, while the computational time stays similar. What explains this situation is that the total number of executors is fixed regardless of the number of *map* tasks, consequently the tasks assigned to a executor have to be executed in a serial way. Another reason is that the available hardware infrastructure is below the Spark hardware requirements.

On the other hand, for MRGA-M and MRGA-H the communication time surpasses the computation time, but it remains stable for every number of map tasks. The reasons of this behavior are the same large dataset size is considered and always the number of maps is equal to the assigned number of cores. However, the computation time becomes smaller when the number of tasks increase because each *map* task is assigned to a different core of the cluster. Moreover, MR-MPI and

Hadoop are better suited to low cost commercial off-the-shelf computers. In the view of these results, we cannot answer *RQ4* clearly. Although the communication time is an important factor to take into account to chose a MRGA solver, this kind of time is strongly related with the number of *maps*, dataset size, and the hardware infrastructure. Therefore, the combination of these last factors could lead or not to a reduction in the communication time.

6 Conclusions

In this article, we have proposed big optimization solvers based on MR implementations of a Simple Genetic Algorithm in different Big Data processing frameworks. These allowed us to solve large instances of combinatorial optimization problems, in particular, the knapsack problem that is important in the industry and academia. In this sense, we used three open-source frameworks as Hadoop, MR-MPI, and Spark in order to generate MRGA-H, MRGA-M, and MRGA-S solvers, respectively. We empirically assessed the effectiveness of the tree MRGA algorithms in terms of execution time, scalability, speedup, and communication vs. computation to answer the research questions formulated at the beginning of this work. This assessment was carried out by using six big instances with sizes varying from 25,000

to 300,000 items, which were chosen to represent different degrees of computational and memory load.

Results show that, from a computational point of view, the execution times of the MRGA solvers increased as the dimensionality of the problem grew, as it was expected. This behavior is exhibited by MRGA-M and MRGA-S, but not by MRGA-H, that is not affected for the instance dimensionality and shows the best speedup values. The MRGA-H then outperforms the other two in terms of execution time when the problem size scales to high dimensional instances. In this way, *RQ1* and *RQ2* are satisfactorily answered.

Furthermore, the answer to the question about the scalability of MRGA solvers to an increased number of *map* task (*RQ3*) was that, in fact, a gain in time is observed if more *map* tasks are used to solve the same problem instance. It is more noticeable for MRGR-M than for the other two MRGA solvers, due to the growing memory requirements as consequence of the increase in the instance sizes. MRGA-S presents the lowest communication times but it can not exploit the advantage to use the in-memory persistence.

However, no conclusive evidence was found with regards to the time spent in communication as a factor to choose a particular MRGA solver, the last research question formulated (*RQ4*) in the present study. The communication time seems to be too much related in an unknown way with the number of *map* tasks, dataset size, and the hardware infrastructure.

The differences observed in the behavior of our MRGA solvers are to some extent explained by the facts that both, MRGA-M and MRGA-S, keep and manage the population from memory, while MRGA-H uses HDFS to manage it. Furthermore, the MRGA-S deserves special attention due to the low performance in its behavior. Given that the population is updated in each iteration, the contents of its RDD persists in memory only one iteration. As a consequence, the Sparks performance advantage with respect to the use in-memory persistence can not be exploited.

Summarizing the above observations, MRGA-H presents a better performance and scalability than MRGA-M and MRGA-S when high dimensional optimization problems are solved. Therefore, the MRGA-H solver continues to perform adequately as its workload grows as much as the the capacity of the containers allows it. Nevertheless, the MRGA-H and MRGA-S should be positively considered since they are using frameworks which allow easier programmability. They also present further advantages, such as inherent support to node failures and data replication.

In a future work, other models to parallelize the SGA, such as the island model, using MR paradigm

will be considered on these three frameworks in order to improve the speedup of the big optimizer and take advantage of the distributed nature of the new proposals. Also, the sensitivity of the parameter settings on the proposed algorithm will be addressed. Other appropriate big optimization problems to analyze the performance of these big optimization solvers will be used to give more insights of their behavior.

Acknowledgments

This research received financial support from the Universidad Nacional de La Pampa and the Incentive Program from MINCyT (Argentina). Moreover, this research has been partially funded by the Spanish MINECO and FEDER project TIN2017-88213-R (6city), and by PRECOG (UMA18-FEDERJA-003).

Conflict of interest

The authors declare that they have no conflict of interest.

Human and animal rights

This article does not contain any studies with animals performed by any of the authors.

Informed consent

Informed consent was obtained from all individual participants included in the study.

References

1. Welcome to Apache Hadoop! Technical report, The Apache Software Foundation, <http://hadoop.apache.org/>, 2014.
2. E. Alba. Parallel evolutionary algorithms can achieve super-linear performance. *Information Processing Letters*, 82(1):7 – 13, 2002. Evolutionary Computation.
3. E. Alba. *Parallel Metaheuristics: A New Class of Algorithms*. Wiley-Interscience, New York, NY, USA, 2005.
4. L. Alterkawi and M. Migliavacca. Parallelism and partitioning in large-scale GAs using spark. In *Proceedings of the Genetic and Evolutionary Computation Conference, GECCO 19*, page 736744, New York, NY, USA, 2019. Association for Computing Machinery.
5. A. Cano, C. García-Martínez, and S. Ventura. Extremely high-dimensional optimization with MapReduce: Scaling functions and algorithm. *Information Sciences*, 415-416(Supplement C):110 – 127, 2017.

6. F. Chávez, F. Fernández, C. Benavides, D. Lanza, J. Villegas, L. Trujillo, G. Olague, and G. Román. ECJ+Hadoop: An easy way to deploy massive runs of evolutionary algorithms. In G. Squillero and P. Burrelli, editors, *Applications of Evolutionary Computation*, pages 91–106, Cham, 2016. Springer International Publishing.
7. J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. In *OSDI04: Proceedings of the 6TH Conference on Symposium on Operating Systems Design and Implementation*. USENIX Association, 2004.
8. F. Ferrucci, P. Salza, and F. Sarro. Using Hadoop MR for parallel GAs: A comparison of the global, grid and island models. *Evol. Computation*, 0(0):1–33, 2017.
9. J. Rui Figueira, G. Tavares, and M. Wiecek. Labeling algorithms for multiple objective integer knapsack problems. *Computers & Operations Research*, 37(4):700 – 711, 2010.
10. M.R. Garey and D.S. Johnson. *Computers and Intractability: a Guide to the Theory of NP-Completeness*. Freeman, 1979.
11. L. Di Geronimo, F. Ferrucci, A. Murolo, and F. Sarro. A parallel genetic algorithm based on Hadoop MapReduce for the automatic generation of JUnit test suites. In *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, pages 785–793, April 2012.
12. D.E. Goldberg. *The Design of Innovation: Lessons from and for Competent Genetic Algorithms*. Kluwer Academic Publishers, 2002.
13. M. Hamstra, H. Karau, M. Zaharia, A. Konwinski, and P. Wendell. *Learning Spark: Lightning-Fast Big Data Analytics*. OReilly Media, 2015.
14. I. Hashem, N. Anuar, A. Gani, I. Yaqoob, F. Xia, and S. Khan. Mapreduce: Review and open challenges. *Scientometrics*, 109(1):389–422, Oct 2016.
15. C. Hu, G. Ren, C. Liu, M. Li, and W. Jie. A spark-based genetic algorithm for sensor placement in large scale drinking water distribution systems. *Cluster Computing*, 20(2):1089–1099, 2017.
16. C. Jatoth, G.R. Gangadharan, U. Fiore, and R. Buyya. Qos-aware big service composition using mapreduce based evolutionary algorithm with guided mutation. *Future Generation Computer Systems*, 86:1008 – 1018, 2018.
17. L. Jenkins. A bicriteria knapsack program for planning remediation of contaminated lightstation sites. *European Journal of Operational Research*, 140(2):427–433, 2002.
18. H. Kellerer, U. Pferschy, and D. Pisinger. *Introduction to NP-Completeness of Knapsack Problems*, pages 483–493. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004.
19. K. Klamroth and M. M. Wiecek. Time-dependent capital budgeting with multiple criteria. In Yacov Y. Haimes and Ralph E. Steuer, editors, *Research and Practice in Multiple Criteria Decision Making*, pages 421–432, Berlin, Heidelberg, 2000. Springer Berlin Heidelberg.
20. M. Lozano, D. Molina, and F. Herrera. Editorial scalability of evolutionary algorithms and other metaheuristics for large-scale continuous optimization problems. *Soft Computing*, 15(11):2085–2087, 2011.
21. B. Miller and D. Goldberg. Genetic algorithms, tournament selection, and the effects of noise. *Complex Systems*, 9:193–212, 1995.
22. C. Paduraru, M. Melemciuc, and A. Stefanescu. A distributed implementation using apache spark of a genetic algorithm applied to test data generation. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, GECCO '17, pages 1857–1863. ACM, 2017.
23. D. Pisinger. Core problems in knapsack algorithms. *Operations Research*, 47:570–575, 1999.
24. S. Plimpton and K. Devine. Mapreduce in MPI for large-scale graph algorithms. *Parallel Computing*, 37(9):610–632, 2011.
25. T. Pradhan, A. Israni, and M. Sharma. Solving the 01 knapsack problem using genetic algorithm and rough set theory. In *2014 IEEE International Conference on Advanced Communications, Control and Computing Technologies*, pages 1120–1125, 2014.
26. R. Qi, Z. Wang, and S. Li. A parallel genetic algorithm based on spark for pairwise test suite generation. *Journal of Computer Science and Technology*, 31:417–427, 2016.
27. V. Quintana Rodriguez and Ma. Laye. Modeling and optimization of content delivery networks with heuristics solutions for the multidimensional knapsack problem. pages 13–18, 2016.
28. A. Salama, M. Wahed, and E. Yousif. Big data flow adjustment using knapsack problem. *Journal of Computer and Communications*, 6:30–39, 2018.
29. C. Salto, G. Minetti, E. Alba, and G. Luque. Developing genetic algorithms using different mapreduce frameworks: MPI vs. Hadoop. In F. Herrera, S. Damas, R. Montes, S. Alonso, Ó. Cordón, A. González, and A. Troncoso, editors, *Advances in Artificial Intelligence*, pages 262–272, Cham, 2018. Springer International Publishing.
30. E. Scott and S. Luke. ECJ at 20: Toward a general metaheuristics toolkit. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, GECCO 19, pages 1391–1398, New York, NY, USA, 2019. Association for Computing Machinery.
31. E. Talbi. *Metaheuristics: From Design to Implementation*. Wiley Publishing, 2009.
32. A. Verma, X. Llorà, D.E. Goldberg, and R. Campbell. Scaling genetic algorithms using MapReduce. In *ISDA'09*, pages 13–18, 2009.
33. A. Verma, X. Llorà, S. Venkataraman, D.E. Goldberg, and R. Campbell. Scaling eCGA model building via data-intensive computing. In *IEEE Congress on Evolutionary Computation*, pages 1–8, 2010.
34. T. White. *Hadoop, The Definitive Guide*. OReilly Media, 2012.
35. M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, pages 2–2. USENIX Association, 2012.
36. Guo Zhou, Ruixin Zhao, and Yongquan Zhou. Solving large-scale 0-1 knapsack problem by the social-spider optimisation algorithm. *IJCSM*, 9(5):433–441, 2018.