Soft Computing (2023) 27:11469–11484
https://doi.org/10.1007/s00500-023-08301-x

OPTIMIZATION

Big optimization with genetic algorithms: Hadoop, Spark, and MPI

Carolina Salto[1,2] · Gabriela Minetti[1] · Enrique Alba[3] · Gabriel Luque[3]

Accepted: 23 April 2023 / Published online: 20 May 2023
© The Author(s), under exclusive licence to Springer-Verlag GmbH Germany, part of Springer Nature 2023

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 study and analyze the MR technology, focusing on Hadoop, Spark, and MPI as the middleware platforms to develop genetic algorithms (GAs). From this, MRGA solvers arise using a different programming paradigm 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 (Lozano et al. 2011). 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 (Cano et al. 2017; Ferrucci et al. 2017; Paduraru et al. 2017; Qi et al. 2016; Verma et al. 2010). 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 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*

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

Carolina Salto
saltoc@ing.unlpam.edu.ar

Enrique Alba
eat@lcc.uma.es

Gabriel Luque
gabriel@lcc.uma.es

1 Facultad de Ingeniería, Universidad Nacional de La Pampa, General Pico, Argentina

2 Conicet, Neuquén, Argentina

3 ITIS Software, Universidad de Málaga, wright, Spain

task (Cano et al. 2017). Hadoop is a very popular framework, relying in the MR paradigm (Welcome 2014; White 2012), both in industry and academia. This framework provides a ready-to-use distributed infrastructure, which is easy to program, scalable, reliable, and fault-tolerant (Hashem et al. Oct 2016). Since Hadoop allows parallelism of data and control, we research for other software tools doing similar jobs. The MapReduce-MPI (MR-MPI) (Plimpton and Devine 2011) 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 (Hamstra et al. 2015) 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 multistep 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 (MRGAs), 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 (Cano et al. 2017; Ferrucci et al. 2017; Di et al. 2012; Verma et al. 2009, 2010) and Spark (Hu et al. 2017; Paduraru et al. 2017; Qi et al. 2016), and a few ones under MR-MPI (Salto et al. 2018), 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 problems 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 incre ased 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) (Goldberg 2002) 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 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 (Salto et al. 2018), coming out from a parallelization of each iteration of a SGA under the iteration-level parallel model (Talbi 2009). The contributions of this work are manyfold. We develop the same optimizer (MRGA) using three open-source MR frameworks. We consider the implementations made in our previous research (Salto et al. 2018), 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 problem size of industrial interest as the knapsack problem (Garey and Johnson 1979). As to our knowledge, this is the first work considering the same MRGA solver implemented in the three widely known platforms and pointing out their different features for the benefit of future researches.

This article is organized as follows: next section discusses the MR paradigm and Big Data frameworks, showing their similarities and differences. Section 3 presents a brief state of the art in implementing GAs with the MR paradigm and contains our proposal. Sections 4 and 5 define meaning-

ful experiments to reveal information on the three systems, perform them, and give some findings. Finally, Sect. 6 summarizes our conclusions and expected future work.

2 MR paradigm and frameworks

An application in the MR paradigm is arranged as a pair (or a sequence of pairs) of *map* and *reduce* functions (Dean and Ghemawat 2004). Each *map* function takes as input a set of key/value pairs (records) from data files and generates a set of intermediate key/value pairs. Then, MR groups together all these intermediate values associated with a same intermediate key. A value group and its associated key are the input to the *reduce* function, which combines these values in order to produce a new and possibly smaller set of key/value pairs that are saved in data files. Furthermore, this function receives the intermediate values via an iterator, allowing the model to handle lists of values that are too large to fit into main memory. The input data are automatically partitioned into a set of M splits when the *map* invocations are distributed across multiple machines, where each input split is processed by a *map* invocation. The intermediate key space is divided into R pieces, which are distributed into R *reduce* invocations. The number of partitions (R) and the partitioning function are user defined.

As previously mentioned, our aim in this work is to perform a comparison of the different Big Data frameworks to develop big optimization MRGA solvers. For that purpose, this section presents three Big Data frameworks, as the Hadoop (Welcome 2014), the MR-MPI (Plimpton and Devine 2011), and Spark (Hamstra et al. 2015), with the goal of identifying the advantages and limitations of each one. In this process, the focus is on the installation, use, and productivity characteristics of each framework.

2.1 Hadoop

The Hadoop framework consists of a single master `ResourceManager`, one slave `NodeManager` per cluster-node, and a `MRAppMaster` per application, which is implemented using the Hadoop YARN framework (Welcome 2014). In order to meet those goals, the central Scheduler (in the `ResourceManager`) responds to a resource request by granting a container. Essentially, the container is the resource 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 tolerance management. In this sense, it is noticeable that the installation and the configuration of the Hadoop framework require a very specific and long sequence of steps, becoming difficult to adapt it to a particular cluster 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.

2.2 MR-MPI

MR-MPI is a small and portable C++ library that only uses MPI for interprocessor communication; thus, the user writes a main program that runs on each processor 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 bandwidth performance and reduce the latency costs.

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 (Zaharia et al. 2012). 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 (returning 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 Spark's 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

Table 1 Comparison between Big Data frameworks

Features	hadoop	MR-MPI	Spark
Open-source	Yes	Yes	Yes
Popular for big-data	Yes	No	Yes
Language supported	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

use with Hadoop. The installation and the configuration of the Spark framework are 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 (Cano et al. 2017; Ferrucci et al. 2017).

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 Verma et al. (2009) and own previous work Salto et al. (2018), 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. (2009, 2010) proposed a SGA and a CGA based on the selecto-recombinative GA, proposed by Goldberg (2002), 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 Di et al. (2012), the authors proposed a similar model of GA than Verma et al. (2009) for software testing. The main difference relays in the use of only one reducer that receives the entire population. Thus, 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. (2017) 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. Chávez et al. (2016) introduce changes in ECJ Scott and Luke (2019) 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. (2018) 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 Paduraru et al. (2017). 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. (2017) use a SGA as optimizer tool, where the population is divided into 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 Alterkawi and Migliavacca (2019) 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 subpopulation. 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 (Ferrucci et al. 2017; Chávez et al. 2016; Jatoth et al. 2018). 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. (2009, 2010), which involves $10^n (n = 4)$ variables and a population of $n \times logn$ size. Finally, Chávez et al. (2016) and Alterkawi and Migliavacca (2019) 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.

Finally, we should comment that this work is based on the preliminary results presented in Salto et al. (2018). However, this paper is a massive update to these results. Firstly, we include a new MRGA solver using SPARK Big Data framework (this is the first work considering the three most known MR frameworks). Secondly, the analysis performed in Salto et al. (2018) was very reduced, mainly a runtime study, since only three cases were tested. This new study considers a comprehensive benchmark with 24 scenarios (eight high-dimensional instances, each one with three different numbers of map tasks). This complete testbed allows us to extensively research several metrics: execution time, scalability, speedup, and communication vs computation, providing a large body of knowledge about Big Data Optimization.

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 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 (Miller and Goldberg 1995). After that, the uniform 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.

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: $Pop'(t)$ = select($Pop(t)$); {k-wise tournament selection without replacement}
6: $offspring$ = recombine($Pop'(t)$, p_c); {uniform crossover}
7: $Pop(t+1)$ = replace($Pop(t)$,$offspring$);
8: evaluate($Pop(t+1)$);
9: $t = t + 1$
10: **end while**
11: return (best individual);

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/multivalue 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 time-consuming 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 whether 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. In this work, due to the absence of the mutation procedure, this disruptive operator is essential to keep a high diversity in the population (Kenneth et al. 1991). 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/multivalue 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 Verma et al. (2009), 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 Verma et al. (2009). The most important ones are related to the changes imposed by passing from the old MRV1 to the new Java MapReduce API MRV2 (Welcome 2014), 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 Fig. 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 Fig. 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 (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 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

Fig. 1 MRGA-H scheme

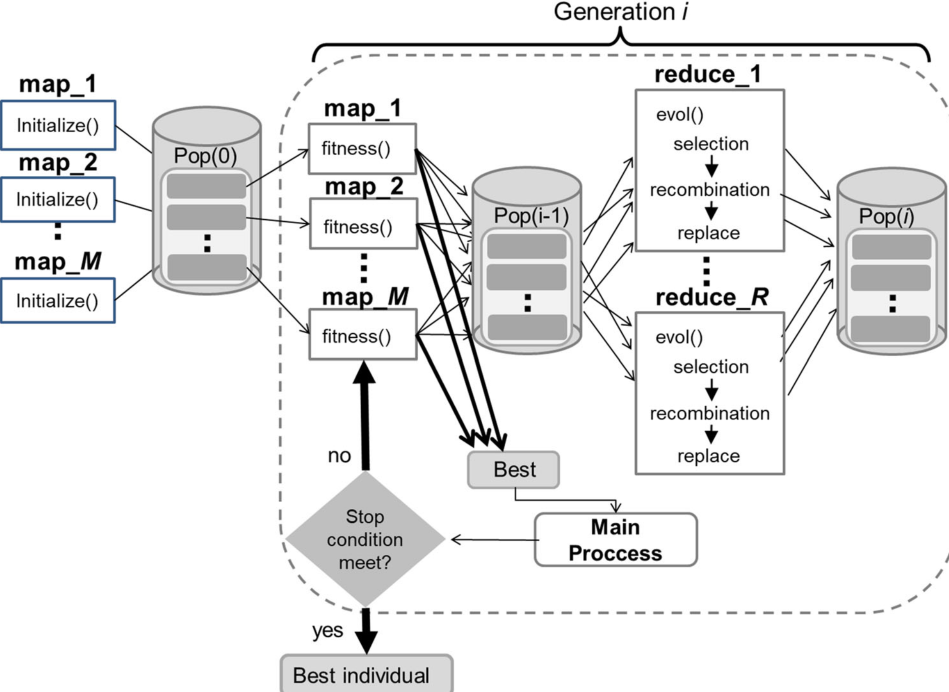
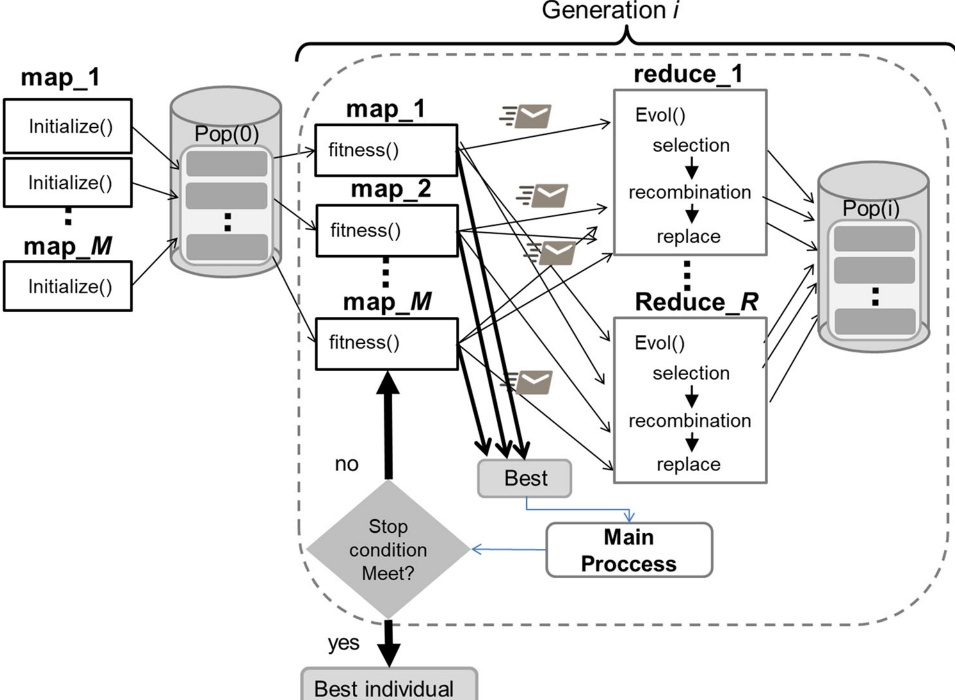


Fig. 2 MRGA-M scheme



key/multivalue pairs. Finally, the $Evol()$ function (from the $reduce$ method) will be called once for each key/multivalue 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.

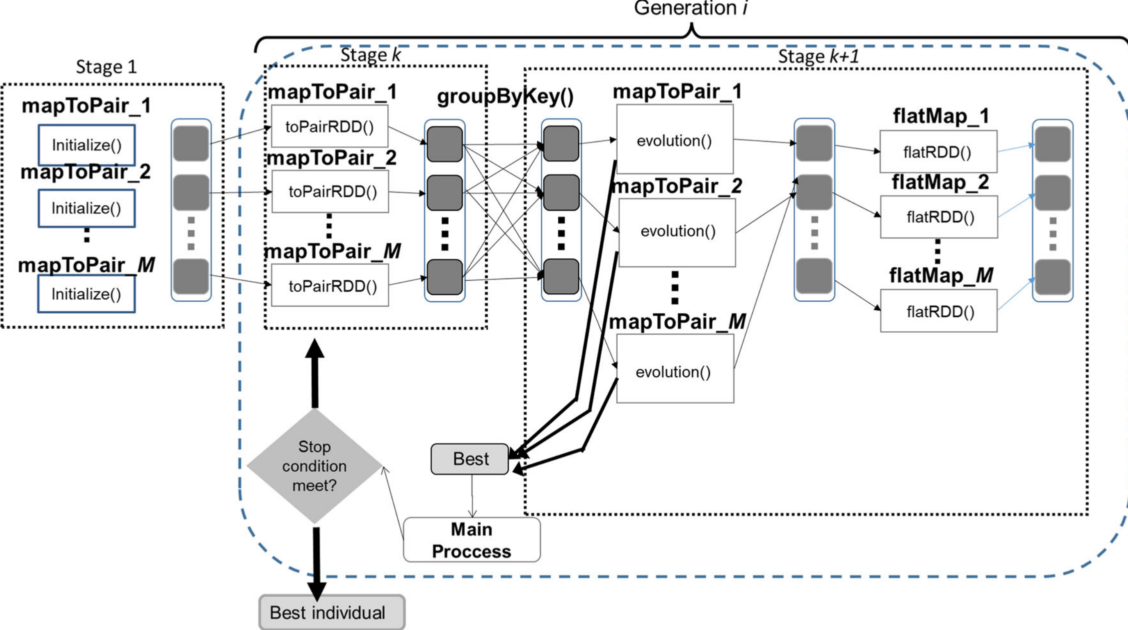


Fig. 3 MRGA-S scheme

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 corresponds 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 $[1, .., R]$, 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 $agregate$() function in MRGA-M.

The next Stage $k + 1$ begins with the redistribution of the individuals across the partitions, by using the *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 Sect. 1, we consider as a benchmark the knapsack problem (Kellerer et al. 2004; Pradhan et al. 2014; Salama et al. 2018; Zhou et al. 2018) 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 0–1 Knapsack Problem (KP) is a classic NP-complete problem (Kellerer et al. 2004), 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 Eq. 1.

$$max \sum_{i=1}^{N_s} x_i \, p_i \tag{1}$$

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 shown in Fig. 4, and its implementation is described in Sect. 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 (Rui Figueira et al. 2010; Jenkins 2002; Klamroth and Wiecek 2000; Quintuna and Laye 2016), 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. To obtain these big KP instances, we use the generator described in Pisinger (1999) and can be found in the repository github.com/GabJL/LargeKPInstances, choosing the uncorrelated data instances type (no correlation between the weight and the profit of an item). We generate 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 knapsack problem.

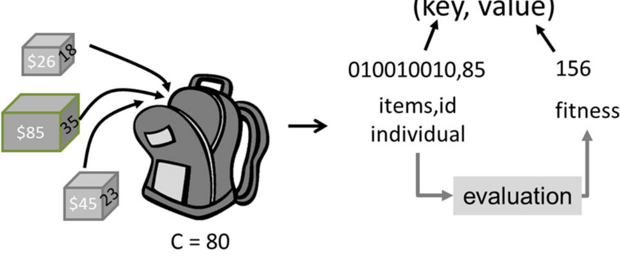


Fig. 4 Solution representation of the knapsack problem in MRGA

4.2 Experimentation methodology

Each MRGA's approach evolves 50,000 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 binary tournament selection to choose 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 vary from 150 MB to 1.8 GB, justifying the use of Big Data frameworks.

Before performing the statistical tests, we first check whether the data follow a normal distribution by applying the Shapiro–Wilks test. Where the data are normally distributed, we later apply an ANOVA test. Otherwise, we use the Kruskal–Wallis (KW) test. These statistical studies allow us to assess whether or not there are meaningful differences between the compared algorithms with $\alpha = 0.05$. These pairwise algorithm differences are determined by conducting a post hoc test, such as the Tukey test for ANOVA or the Wilcoxon test for the KW one.

4.3 Evaluation metrics

In the previous section, we explained the methodology for experimentation, and 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, mean profit values, and speedup. This becomes as a good practice to report results in the metaheuristic field (Alba 2005). 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.

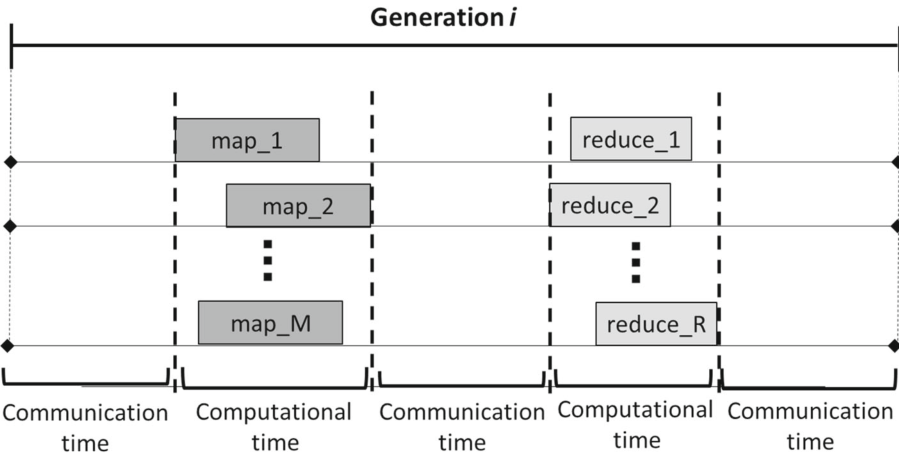


Fig. 5 The method to compute times for MRGA-M and MRGA-H

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 Alba (2002) 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 Alba's taxonomy (Alba 2002).

Communication vs. Computation. A study about the communication and computation times of each algorithm allows us to understand the reasons that causes our proposals have a slightly improved speedup. We adopt the method used in Ferrucci et al. (2017), 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 online and run each algorithm (communication time).

Mean Profit. The mean profit values consider the average of the best solutions found by each MRGA solver to evaluate their efficacy.

5 Result analysis

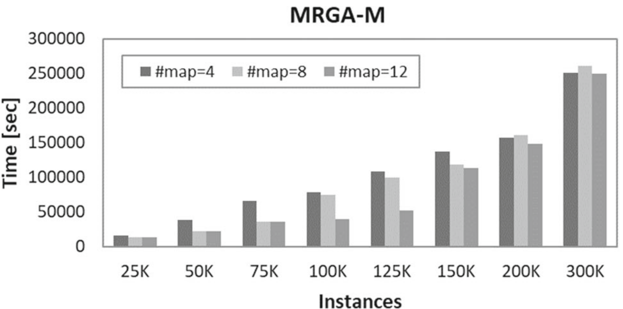
In this section, we present the results that allows us to answer the different *RQs* formulated in Sect. 1. The comparison between the MRGA solvers with respect to the scalability is in Sect. 5.1. The analysis of the speedup is in Sect. 5.2. In Sect. 5.3, we contrast the communication and computation times consumed by each MRGA. Finally, the analysis of mean profit values is presented in Sect. 5.4.

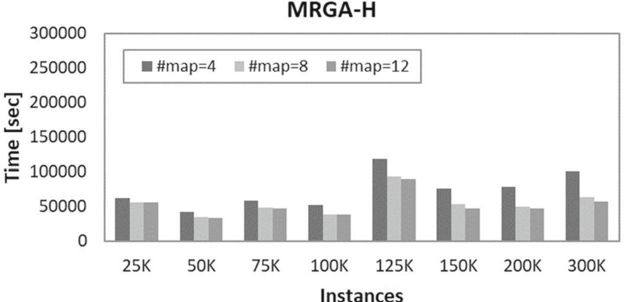
5.1 Scalability

Table 2 and Fig. 6 show the execution times achieved for each algorithm by increasing the problem dimension. Furthermore, in Table 2, the respective standard deviation values ($\pm SD$) are introduced. In the last column of each sub-table, the results of the KW test are summarized, where the symbol "+" indicates that the execution times of the MRGA solvers are statistically similar, while the symbol "-" specifies these times are significantly different. It is noticeable that the MRGA-S execution time for the 300K instance is not available (N/A) because this solver cannot execute such a large instance in our systems. For every instance, each bar of Fig. 6 represents a different number of *map* tasks. As expected, the execution times increase as the problem's dimensional-

Table 2 Mean execution times achieved for each MRGA solver and its respective standard deviation (SD)

Inst.	MRGA-M			
	#maps=4	#maps=8	#maps=12	KW
25K	15,420$_{\pm 38}$	14,025$_{\pm 25}$	14,130$_{\pm 10}$	+
50K	39,039$_{\pm 60}$	22,424$_{\pm 25}$	22,710$_{\pm 20}$	−
75K	66,585$_{\pm 141}$	36,600$_{\pm 21}$	36,570$_{\pm 16}$	−
100K	78,894$_{\pm 177}$	74,724$_{\pm 19}$	40,150$_{\pm 17}$	−
125K	108,645$_{\pm 449}$	99,390$_{\pm 36}$	51,810$_{\pm 16}$	−
150K	136,815$_{\pm 349}$	118,110$_{\pm 56}$	113,798$_{\pm 25}$	−
200K	157,475$_{\pm 270}$	160,672$_{\pm 49}$	148,570$_{\pm 17}$	+
300K	251,028$_{\pm 422}$	260,218$_{\pm 62}$	248,944$_{\pm 29}$	+
Inst.	MRGA-H			
	#maps=4	#maps=8	#maps=12	KW
25K	62,862 $_{\pm 150}$	56,798$_{\pm 99}$	55,979$_{\pm 39}$	+
50K	42,607 $_{\pm 164}$	35,256$_{\pm 68}$	33,931$_{\pm 54}$	−
75K	58,440 $_{\pm 378}$	48,222$_{\pm 55}$	47,598$_{\pm 42}$	−
100K	52,235 $_{\pm 394}$	38,542$_{\pm 43}$	38,479$_{\pm 37}$	−
125K	118,171$_{\pm 894}$	94,093$_{\pm 72}$	89,665$_{\pm 32}$	−
150K	75,932 $_{\pm 653}$	54,267$_{\pm 105}$	48,066$_{\pm 47}$	−
200K	79,271 $_{\pm 523}$	50,402$_{\pm 95}$	47,387$_{\pm 33}$	−
300K	101,026$_{\pm 819}$	63,380$_{\pm 120}$	57,806$_{\pm 57}$	−
Inst.	MRGA-S			
	#maps=4	#maps=8	#maps=12	KW
25K	22,952$_{\pm 58}$	19,391$_{\pm 26}$	23,349$_{\pm 49}$	+
50K	38,108$_{\pm 132}$	30,232$_{\pm 199}$	28,033$_{\pm 178}$	−
75K	51,448$_{\pm 357}$	40,956$_{\pm 369}$	37,335$_{\pm 321}$	−
100K	65,416$_{\pm 465}$	50,923$_{\pm 404}$	45,182$_{\pm 376}$	−
125K	79,799$_{\pm 397}$	61,932$_{\pm 389}$	55,904$_{\pm 301}$	−
150K	94,052$_{\pm 528}$	73,351$_{\pm 471}$	66,071$_{\pm 485}$	−
200K	123,655$_{\pm 897}$	71,088$_{\pm 754}$	83,443$_{\pm 768}$	−
300K	N/A	172,853$_{\pm 1023}$	208,043$_{\pm 548}$	−





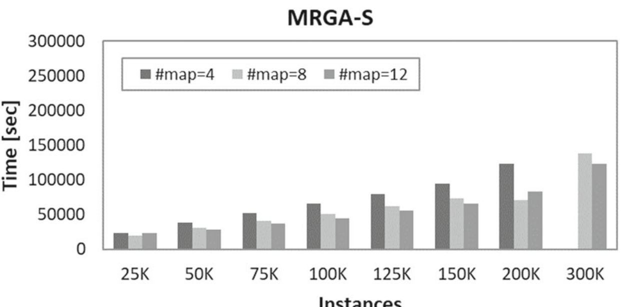


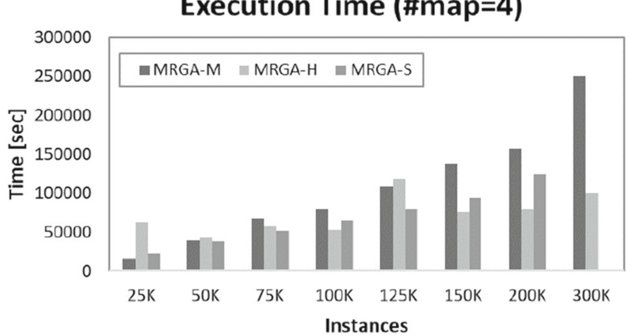
Fig. 6 Mean execution time of MRGA algorithms to solve the KP instances

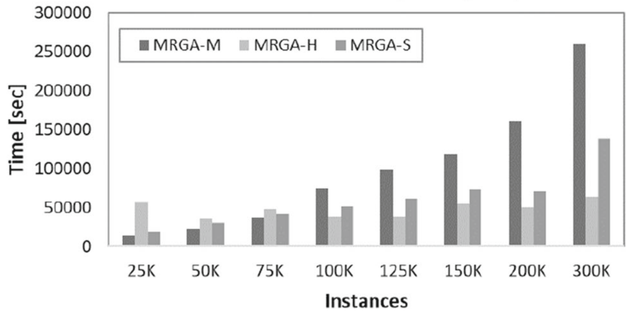
ity grows. This situation is evident in the case of MRGA-M and MRGA-S. However, MRGA-H presents a behavior without direct dependence on the problem size. The previous results allow us to answer the *RQ2* since these algorithms can efficiently solve incremental high-dimensional instances, becoming scalable 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 Fig. 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 obtain a gain in the time. The KW results statistically support these differences. 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.

5.2 Speedup

We analyze the results shown in Fig. 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





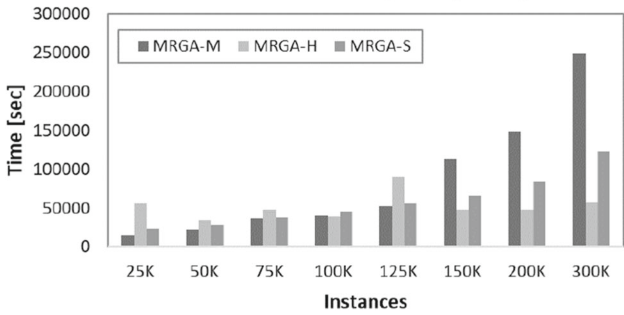


Fig. 7 Mean execution time of MRGA algorithms for each number of map tasks to solve the KP instances

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 Fig. 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 Fig. 9.

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 an 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 to 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.

5.4 Mean profit

Figure 11 shows the mean profit values obtained by each MRGA solver for each instance. We empirically and statistically verified that no significant differences in the MRGA efficacy are observed. These are expected results because the

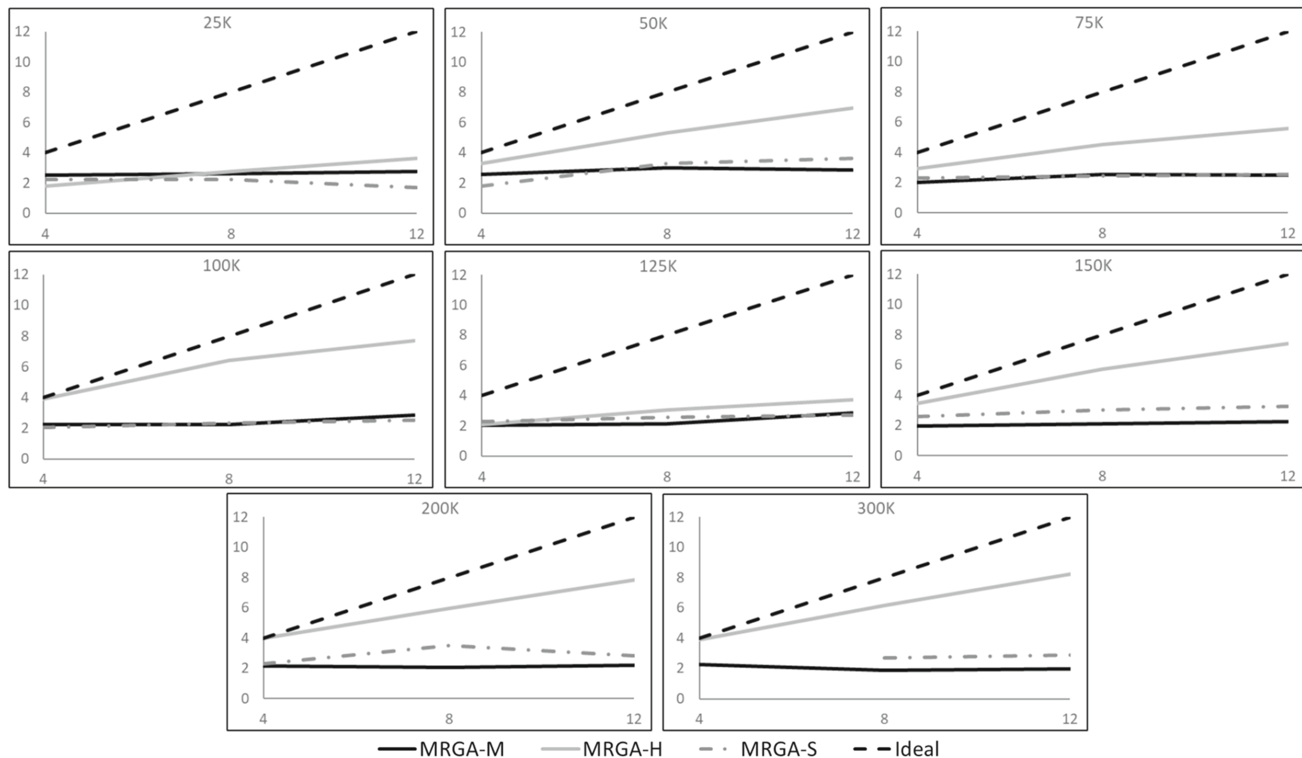


Fig. 8 Speedup trend per instance

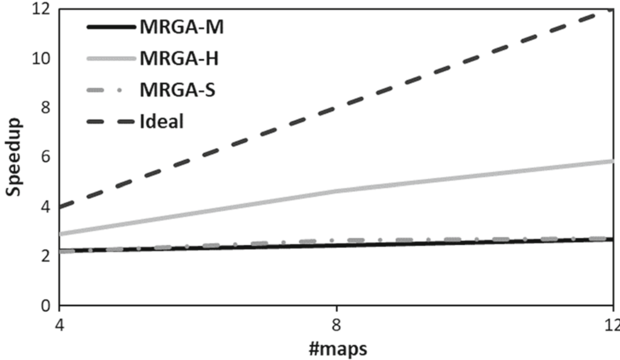


Fig. 9 Mean speedup per MRGA solver

primary goal of MRGA solvers is to preserve the SGA behavior while enabling it to tackle large-scale problem instances.

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 three 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 a consequence of the increase in the instance sizes. MRGA-S presents the lowest communication times, but it cannot exploit the advantage to use the in-memory persistence.

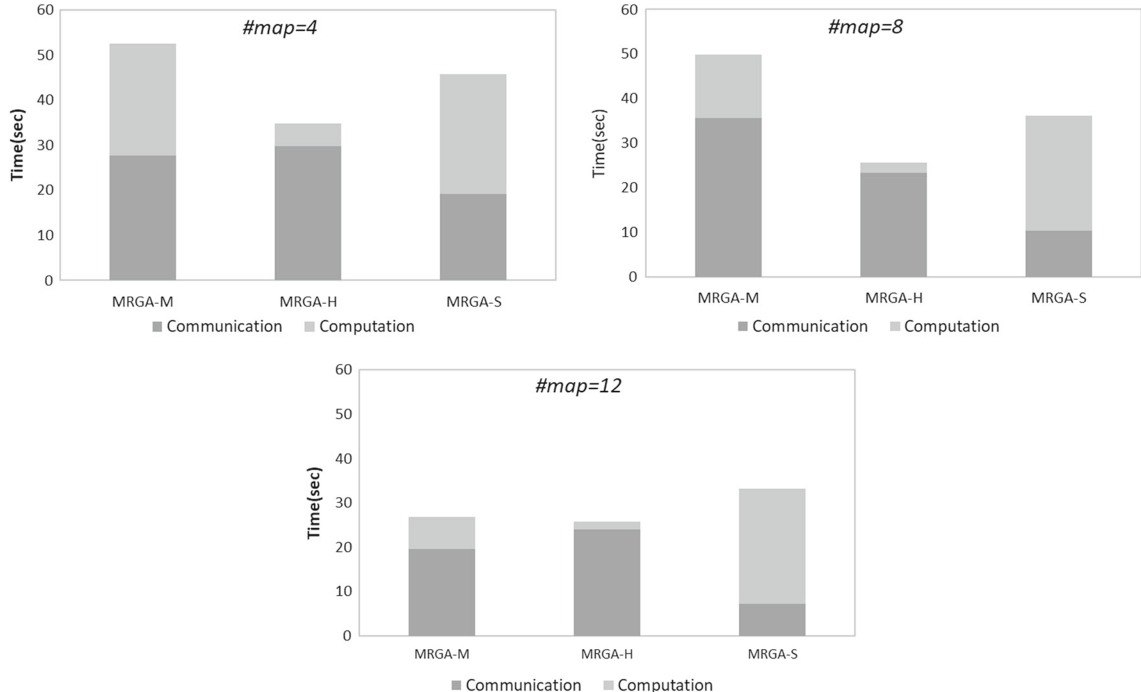


Fig. 10 The average communication vs. computational times spent by each algorithm per generation

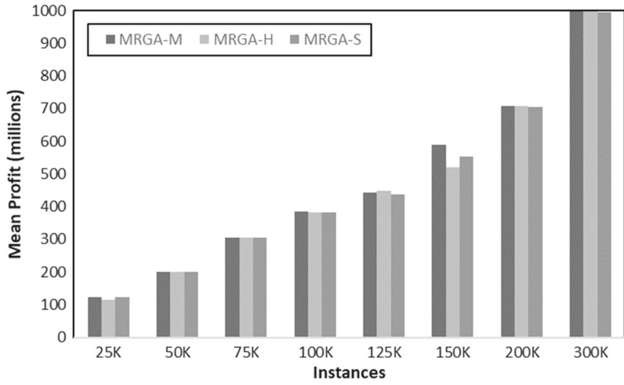


Fig. 11 Mean profit values per MRGA solver

However, no conclusive evidence was found with regard 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 Spark's performance advantage with respect to the use in-memory persistence cannot 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 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.

Funding This research received financial support from the Universidad Nacional de La Pampa and the Incentive Program from MINCyT (Argentina). Moreover, this research is partially funded by the Universidad de Malaga; under grant PID 2020-116727RB-I00 (HUmove)

funded by MCIN/AEI/10.13039/501100011033; and TAILOR ICT-48 Network (No 952215) funded by EU Horizon 2020 research and innovation programme.

Data availibility Enquiries about data availability should be directed to the authors.

Declarations

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

Alba E (2002) Parallel evolutionary algorithms can achieve super-linear performance. Inf Process Lett 82(1):7–13

Alba E (2005) Parallel metaheuristics: a new class of algorithms. Wiley-Interscience, New York

Alterkawi L, Migliavacca M (2019) Parallelism and partitioning in large-scale GAs using spark. In: Proceedings of the genetic and evolutionary computation conference, GECCO'19. New York, NY, USA. Association for Computing Machinery, pp 736–744

Cano A, García-Martínez C, Ventura S (2017) Extremely high-dimensional optimization with MapReduce: scaling functions and algorithm. Inf Sci 415, 416(Supplement C):110–127

Chávez F, Fernández F, Benavides C, Lanza D, Villegas J, Trujillo L, Olague G, Román G (2016) ECJ+Hadoop: an easy way to deploy massive runs of evolutionary algorithms. In: Squillero G, Burelli P (eds) Applications of evolutionary computation. Springer, Cham, pp 91–106

De Kenneth J, William S (1991) An analysis of the interacting roles of population size and crossover in genetic algorithms. Parallel Problem Solv Nat 1:38–47

Dean J, Ghemawat S (2004) MapReduce: simplified data processing on large clusters. In: OSDI'04: proceedings of the 6TH conference on symposium on operating systems design and implementation. USENIX Association

Di L, Geronimo, Ferrucci F, Murolo A, Sarro F (2012) 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, April 2012. pp 785–793

Ferrucci F, Salza P, Sarro F (2017) Using Hadoop MR for parallel GAs: a comparison of the global, grid and island models. Evol Comput. https://doi.org/10.1162/evco_a_00213

Garey MR, Johnson DS (1979) Computers and intractability: a guide to the theory of NP-completeness. Freeman, San Francisco

Goldberg DE (2002) The design of innovation: lessons from and for competent genetic algorithms. Kluwer, Boston

Guo Z, Ruixin Z, Yongquan Z (2018) Solving large-scale 0–1 knapsack problem by the social-spider optimisation algorithm. IJCSM 9(5):433–441

Hamstra M, Karau H, Zaharia M, Konwinski A, Wendell P (2015) Learning spark: lightning-fast big data analytics. OReilly Media, Sebastopol

Hashem I, Anuar N, Gani A, Yaqoob I, Xia F, Khan S (2016) Mapreduce: review and open challenges. Scientometrics 109(1):389–422

Hu C, Ren G, Liu C, Li M, Jie W (2017) A spark-based genetic algorithm for sensor placement in large scale drinking water distribution systems. Clust Comput 20(2):1089–1099

Jatoth C, Gangadharan GR, Fiore U, Buyya R (2018) QoS-aware big service composition using mapreduce based evolutionary algorithm with guided mutation. Futur Gener Comput Syst 86:1008–1018

Jenkins L (2002) A bicriteria knapsack program for planning remediation of contaminated lightstation sites. Eur J Oper Res 140(2):427–433

Kellerer H, Pferschy U, Pisinger D (2004) Introduction to NP-completeness of knapsack problems. Springer, Berlin, pp 483–493

Klamroth K, Wiecek MM (2000) Time-dependent capital budgeting with multiple criteria. In: Haimes YY, Steuer RE (eds) Research and practice in multiple criteria decision making. Springer, Berlin, pp 421–432

Lozano M, Molina D, Herrera F (2011) Editorial scalability of evolutionary algorithms and other metaheuristics for large-scale continuous optimization problems. Soft Comput 15(11):2085–2087

Miller B, Goldberg D (1995) Genetic algorithms, tournament selection, and the effects of noise. Complex Syst 9:193–212

Paduraru C, Melemciuc M, Stefanescu A (2017) 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. ACM, pp 1857–1863

Pisinger D (1999) Core problems in knapsack algorithms. Oper Res 47:570–575

Plimpton S, Devine K (2011) Mapreduce in MPI for large-scale graph algorithms. Parallel Comput 37(9):610–632

Pradhan T, Israni A, Sharma M (2014) Solving the 0–1 knapsack problem using genetic algorithm and rough set theory. In: 2014 IEEE international conference on advanced communications, control and computing technologies. pp 1120–112

Qi R, Wang Z, Li S (2016) A parallel genetic algorithm based on spark for pairwise test suite generation. J Comput Sci Technol 31:417–427

Quintuna RV, Laye M (2016) Modeling and optimization of content delivery networks with heuristics solutions for the multidimensional knapsack problem. pp 13–18

Rui Figueira J, Tavares G, Wiecek M (2010) Labeling algorithms for multiple objective integer knapsack problems. Comput Oper Res 37(4):700–711

Salama A, Wahed M, Yousif E (2018) Big data flow adjustment using knapsack problem. J Comput Commun 6:30–39

Salto C, Minetti G, Alba E, Luque G (2018) Developing genetic algorithms using different mapreduce frameworks: MPI vs. Hadoop. In: Herrera F, Damas S, Montes R, Alonso S, Cordón Ó, González A, Troncoso A (eds) Advances in artificial intelligence. Springer, Cham, pp 262–272

Scott E, Luke S (2019) ECJ at 20: Toward a general metaheuristics toolkit. In: Proceedings of the genetic and evolutionary computation conference companion, GECCO'19, New York, Association for Computing Machinery, pp 1391–1398

Talbi E (2009) Metaheuristics: from design to implementation. Wiley, New York

Verma A, Llorà X, Goldberg DE, Campbell R (2009) Scaling genetic algorithms using MapReduce. In: ISDA'09, pp 13–18

Verma A, Llorà X, Venkataraman S, Goldberg DE, Campbell R (2010) Scaling eCGA model building via data-intensive computing. In: IEEE congress on evolutionary computation, pp 1–8

Welcome to (2014) Apache™ Hadoop®! Technical report. The Apache Software Foundation. http://hadoop.apache.org/

White T (2012) Hadoop, the definitive guide. O'Reilly Media, Sebastopol

Zaharia M, Chowdhury M, Das T, Dave A, Ma J, McCauleyM, Franklin M, Shenker S, Stoica I (2012) 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. USENIX Association, pp 2–2

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Springer Nature or its licensor (e.g. a society or other partner) holds exclusive rights to this article under a publishing agreement with the author(s) or other rightsholder(s); author self-archiving of the accepted manuscript version of this article is solely governed by the terms of such publishing agreement and applicable law.