Future Generation Computer Systems 92 (2019) 276–289

Contents lists available at ScienceDirect

Future Generation Computer Systems

journal homepage: www.elsevier.com/locate/fgcs

Speed up genetic algorithms in the cloud using software containers

Pasquale Salza [a,*], Filomena Ferrucci [b]

[a] Faculty of Informatics, USI Università della Svizzera italiana, Lugano, Switzerland
[b] Department of Computer Science, University of Salerno, Italy

HIGHLIGHTS

- An approach to deploy distributed Genetic Algorithms (GAs) in the cloud.
- A software engineering workflow to develop, deploy and execute distributed GAs.
- Its effectiveness is measured in execution time, speedup, overhead and setup time.
- A comparison with the state-of-the-art approaches, highlighting the pros and cons.

ARTICLE INFO

Article history:
Received 25 October 2017
Received in revised form 12 September 2018
Accepted 30 September 2018
Available online xxxx

Keywords:
Genetic algorithms
Parallel genetic algorithms
Cloud computing
Software containers
Software engineering

ABSTRACT

Scalability issues might prevent Genetic Algorithms from being applied to real world problems. Exploiting parallelisation in the cloud might be an affordable approach to getting time efficient solutions that benefit of the appealing features of scalability, resource discovery, reliability, fault-tolerance and cost-effectiveness. Nevertheless, parallel computation is very prone to cause considerable overhead for communication. Also, making Genetic Algorithms distributed in an on-demand fashion is not trivial. Aiming at keeping under control the communication cost and, at the same time, supporting developers in the construction and deployment of parallel Genetic Algorithms, we designed and implemented a novel approach to distribute Genetic Algorithms in the form of a cloud-based application. It is based on the master/slave model, exploiting software containers, their cloud orchestration and message queues. We also devised a conceptual workflow covering each cloud Genetic Algorithms distribution phase, from resources allocation to actual deployment and execution, in an engineered fashion. Then, the application performance has been evaluated using a benchmark problem. According to the performance and setup times results, it emerged that the cloud can be considered a compelling way of scaling Genetic Algorithms and an excellent alternative to other technologies strictly related to the physically owned hardware.

1. Introduction

GAs are a powerful technique used in many different fields to search for a near-optimal solution when searching for the optimum is too expensive. Although attractive and elegant in the laboratory, scalability issues prevent GAs from being effectively applied to real world problems [1]. However, parallelisation may be a suitable way to improve the computational time and the effectiveness in the exploration of the search space. Indeed, GAs are 'naturally parallelisable', for instance, their population-based characteristics allow us to evaluate in a parallel way the fitness of each individual, i.e., the 'global parallelisation model', also known as the 'master/slave model' [2].

It is argued that a barrier to the wider application of parallel execution has been the high cost of parallel architectures and infrastructures and their management. GAs have been effectively parallelised on multi-core (i.e., CPUs) and many-core (i.e., GPUs) systems [3,4]. However, these solutions are often expensive and may obtain only a certain degree of parallelisation being strictly related to the number of multiple computational units available on the hardware. On the contrary, technologies based on network communication may hypothetically be scaled without limits, e.g., grid computing [5,6]. *Cloud computing* can represent an affordable solution to address the above issues because it breaks the barrier between employed resources and costs: in a short time, it is possible to allocate a cluster of the desired size without investing in expensive local hardware and its management [7,8].

Previous proposals for distributed GAs in the cloud exploited well-known technologies such as Hadoop MapReduce [9–12], some of them also providing framework/libraries [9,10,13,14] to support developers in building distributed GAs. Even though Hadoop offers some appealing features, the data exchange through a distributed file system may slow down the execution of parallel GAs [11,12]. Moreover, it is required to have dedicate skills for

* Corresponding author.
E-mail addresses: pasquale.salza@usi.ch (P. Salza), fferrucci@unisa.it (F. Ferrucci).

https://doi.org/10.1016/j.future.2018.09.066
0167-739X/

setup and maintenance activities, which often cannot be automatised, to have a fully operational cluster.

Based on these considerations, in this paper we present *AMQPGA*, a novel approach to distributing GAs, implementing a *Cloud-Based Application* (CBA) [15] based on the master/slave model and exploiting technologies especially devised for the cloud (i.e., Docker, CoreOS and RabbitMQ) to fully take advantage of the appealing features of orchestration [16], resource discovery, fault-tolerance, scalability and performance optimisation. It also allows GAs developers to use existing implementations of genetic operators or external tools, without constraints on the adopted programming languages. Indeed, 'software containers' (i.e., Docker containers) provide isolated environments (i.e., virtual Linux instances) where developers can include everything is needed for the computation [17]. We propose a conceptual workflow to support the development, deployment and execution of distributed GAs, exploiting modern software engineering methodologies and tools, e.g., *Continuous Integration* (CI) and *Continuous Deployment* (CD), aiming at reducing the human effort. Moreover, by means of the employed technologies, we made the application surpass the limitations of the number of machines the cloud providers usually impose upon their users, through an infrastructure definable as 'multi-cloud', i.e., based on the allocation of instances by different cloud providers but participating in the same application [18].

The main contributions of this paper can be summarised as follows:

- a conceptual workflow describing the phases of development, deployment and execution of distributed GAs, using modern software engineering methodologies and tools;
- an approach to deploy containers of distributed GAs applications in cloud environments, by implementing the master/slave model and exploiting message queues;
- an empirical study to assess the effectiveness of our application measuring the execution time, speedup, overhead and setup time;
- a comparison with the state-of-the-art approaches, highlighting the pros and cons.

The source code is shared at the address https://github.com/pasqualesalza/amqpga under the terms of the *MIT License*.

The rest of the paper is organised as follows. Section 3 describes some relevant related work. In Section 2 we motivate the need of applying software engineering methodologies to GAs, illustrating the conceptual workflow we devised for the development, deployment and execution of GAs in cloud environments. In Section 4 we summarise the main features of the employed cloud technologies whereas in Section 5 we present the proposed approach. Section 6 and Section 7 report, respectively, the design and the results of the empirical study we carried out to assess the effectiveness of the proposed approach. In Section 8 we compare the AMQPGA approach with the-state-of-art tools for GA parallelisation. Section 9 concludes with some final remarks and future work.

2. Genetic algorithms engineering

GAs have been applied to many fields [19,20]. They revealed themselves to be effective in particular when the problem to solve is computationally challenging. Instead of using traditional methods, the problem can be solved starting from the solutions, i.e., the individuals, to which several genetic operators are applied generation by generation, aiming at improving the quality of the entire population. The heart of GAs is the fitness function, which allows evaluating, discriminating and classifying the individuals. Therefore, obtaining a better solution means optimising the fitness function values. Even though GAs, as any Search-Based algorithms, are already used to considerably reduce the time required to obtain a proper solution to a problem compared to traditional methods, the fitness function and other operators can be very time-consuming components for the whole computation. For this reason, parallelisation can be used as a way to reduce the execution time. However, the GA requires to be explicitly adapted to be parallelised [14]. Thus, GAs might result in being a complex software system to implement, both in terms of systematic development, deployment and execution, which in the case of this paper exploits the cloud.

In this section, we first describe the workflow model we devised to engineer a GA, i.e., transforming the GA into a *Cloud-Based Application* (CBA) [15]. We involve modern software engineering methodologies and tools, allowing the programmers to develop GAs without any technological limits, motivating the use of software containers. Then, we discuss the possibility of connecting the systematic development process to automatic deployment, using the cloud as a ready to go, crowdfundable and parallel platform.

2.1. Develop a genetic algorithm

Develop a GA consists of two main key ingredients [19,21,22]: 1. choose a representation of the problem; 2. define the genetic operators.

First of all, the problem has to be expressed in terms of its solutions, called 'individuals' in this context. In order to be treated, every individual has to be encoded with a gene representation. For instance, let us assume to want to solve the 'Knapsack problem', where the aim is to optimise the selection of items, each having a specific size and value, for a knapsack with a limited capacity. A solution to the problem is then any possible selection of items. The best solution is one where the total value of the knapsack is the maximum obtainable. A possible representation is a binary encoding for which each bit in a string says if a corresponding indexed item will be included in the knapsack.

Once defined the encoding, it is needed to define the genetic operators. Some of them are generic and can be easily applied to any problem, e.g., selection. Others have to be adapted to the specific problem since they work accordingly to determined representations, and their behaviour may cause the production of inconsistent solutions. In particular, the 'fitness evaluation' function characterises the problem. It allows evaluating how much an individual fits the problem by associating a value, i.e., single-objective fitness function, or multiple values, i.e., multi-objective function. The associated values must belong to a partially ordered set ('poset'), thus being comparable and sortable, e.g., real numbers, letting determine if an individual is better than another. Then, other genetic operators can be applied.

Except for the characteristics above mentioned, many of the required components can be repeated to be used to solve different problems. For this reason, many frameworks and libraries for GAs are available [23–26]. They provide ready to use utilities that only need to be customised, let developers focus on the adaptation of the problem. Besides some effort can be saved by using these utilities, GAs are not different from other kinds of software in terms of development. They can consist of complex pieces of code, which might be developed by different programmers, tested and maintained over time.

Fig. 1 depicts a conceptual workflow we devised for a possible real world scenario in which the development, deployment and execution of distributed GAs are performed in an engineered fashion. Here we analyse the workflow from the perspective of the developer. The developer programs his/her GA and, to be sure to keep track of the changes in the source code, can exploit the power of *Version Control Systems* (VCSs). VCSs, e.g., Git, Subversion, are software tools that help developers to back up the source code, allowing collaboration [27]. Besides keeping a local copy,

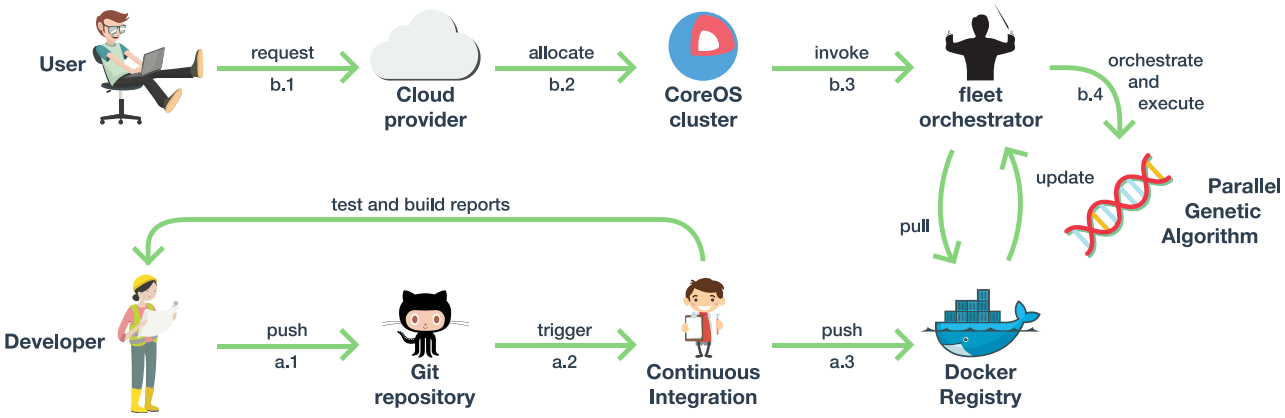

Fig. 1. GAs development, deployment and execution workflow.

developers can share a remote repository for source code where the changes are propagated to the whole team. From the point of view of GAs developers, it lets follow the evolution of the GA software over time, allowing to revert to previous versions if needed. Moreover, multiple developers can collaborate simultaneously in the definition of the problem in terms of encoding and genetic operators. VCSs are also a perfect way to distribute knowledge since it is possible to make available the source code on public remote repositories, where potentially any researcher can collaborate.

In Fig. 1.a.1, the developer uploads, i.e., pushes, the source code on a remote repository. To maintain GAs, a developer cannot leave out from consideration a test phase. Not only does s/he need to test single genetic operators, but also the entire GA might require to be executed against a test suite. Being GAs very time-consuming, *Continuous Integration* (CI) tools are particularly indicated. CI is the practice of frequently integrate shared code in a common repository [28]. Each integration is verified by executing automated testing. There exist several CI platforms that can be automatically remotely triggered (Fig. 1.a.2) as soon as a new version of the code is available on a VCS repository. They usually reproduce a proper environment for the test suite, perform the test execution and collect reports for the developers. The sets of operations can be run independently of the development activity, even on a remote machine.

2.2. Deploy and execute a genetic algorithm

As mentioned before, the fitness evaluation function is the core part of GAs and it usually the most time-consuming component of the whole execution. Cloud computing can be a good option to parallelise GAs, both in terms of efficiency and cost [8]. It is not needed to own any physical infrastructure since computation can be purchased as a service from cloud providers, in the form of virtual instances, for the desired time, quantity and quality. It offers the 'scalability' feature, i.e., the capability of enlarging the number of computational units on demand. Moreover, it includes many mechanisms to guarantee fault tolerance in case of physical and logical failures.

Recently, the architecture of containers is becoming a standard way to distribute applications easily over the cloud. Differently from the traditional hypervisor-based virtualisation, software containers share the common resources of the underlying hardware or virtual instance and Linux kernel, without impacting on performance. Related to GAs, not only can software containers improve execution by parallelising them, but also they can be exploited to define an ad hoc environment. Often, the fitness function consists of the execution of a particular and time-consuming external program with which the GA may need to interact. With containers, the developer can define both the environment for the execution of genetic operators and any required external tools.

Several important companies widely make use of native platforms to manage containers on their infrastructures, e.g., CoreOS, Docker Swarm, Kubernetes. Thousands of containers are spread over their hardware to balance the computational load and guarantee reliability. These platforms give the feeling of having multiple resources as a single pool of computational power. The users can submit a task, e.g., the execution of a container, and the platforms will automatically manage which resources using. The resource does not even need to belong to the same geographical place or cloud provider, i.e., 'multi-cloud' [18], since several resources can be aggregated to the same cluster by running the same platform. It also opens to the concept of 'crowdfunding', for which multiple users contribute by sharing some machine they have commissioned from their own account with their cloud provider. In particular, we exploited CoreOS to this purpose for the proposed implementation (see Section 4.3).

We refer again to Fig. 1, this time from the perspective of the end user who intends to run a parallel GA in the cloud. The end user might or not coincide with the original developer. Let us suppose that a GA is available on a Git source code repository. Besides allowing automatic testing, CI tools also allow to build software automatically. Once the CI process has been triggered (Fig. 1.a.2), the activity of the building can be performed. As for the GAs, it corresponds to the preparation of the environment, compiling the source code and packaging it in the form of a container image (see Section 4.2). This binary representation is then stored on a private or public repository (Fig. 1.a.3), which is therefore reachable from the outside, ready to be downloaded and executed on local machines and in the cloud. This process is commonly known as *Continuous Deployment* (CD), where the software is deployed or released as soon as it is ready [29].

At this point, the end user may decide to run the container on his/her laptop machine or in the cloud. First, s/he requests for virtual machine instances to a cloud provider (Fig. 1.b.1). The next step is to allocate a platform to manage containers, e.g., CoreOS (see Section 4.3) (Fig. 1.b.2). Once configured and invoked (Fig. 1.b.3), the orchestrator can start to perform (Fig. 1.b.4): 1. the download of the most updated version of the GA container image, 2. the scheduling, i.e., orchestration, of containers in the cloud cluster. Cloud orchestration platforms manage everything concerned networking, in particular, the 'discovery' of resources. It is not needed to know a priori which are the *Internet Protocol* (IP) addresses of containers in the cloud, but they collect this dynamic information in a distributed storage that is accessible from any point of the network. Moreover, usually they use names instead of IP addresses, using a *Domain Name System* (DNS) that translates

to physical locations. Also, it allows hiding under the same name multiple instances of the same service. In this way, a load balancer can decide which resource is less busy to accept a new request. Of course, even if very powerful, this capability needs to be specifically designed for the developed CBA. Regarding the scalability, this allows adding an undetermined number of resources to the computation, as in the case of the proposed CBA for GAs.

Therefore, the parallel GA can be finally executed. How the several containers interact is addressed in the next section, in which the application we devised is proposed.

3. Related work

A wide range of work is present in the literature about models and technologies for GAs parallelisation [2]. However, our work aims to parallelise GAs on a commercial cloud environment. Thus, we only report the most relevant work involving models, technologies, problems and conceptual deployment workflows in the GAs or *Evolutionary Algorithms* (EAs) fields.

Zheng et al. compared the multi-core (i.e., CPUs) and the many-core (i.e., GPUs) systems for GAs parallelisation [3]. Firstly, they found that the system based on GPUs is faster than the CPUs one. However, they observed that an architecture with a fixed number of parallel participants, such as GPU cores, might perform worse in terms of quality of solutions than another with more parallel nodes stating that distributed architectures, e.g., the cloud, are worth for GAs parallelisation.

Many authors used the *MapReduce* paradigm to implement parallel GAs [30,31] and some of them with *Hadoop MapReduce* in particular [11,12]. On the one hand, they claimed that GAs can efficiently scale on multiple Hadoop nodes. On the other hand, they highlighted the worrying presence of overhead, due to the communication with the data store, i.e., *Hadoop Distributed File System* (HDFS), suggesting their use only with large populations and intensive computation work for fitness evaluation. In general, Hadoop MapReduce represents one of the most mature and employed technologies to develop parallel algorithms since it provides a ready to use distributed infrastructure that is scalable, reliable and fault-tolerant [32]. Nevertheless, it requires high performance from the underlying hardware. Moreover, the scalability of the infrastructure is possible, but it requires a considerable amount of time before a new node becomes available and it is not suitable for all since specific skills for setup and maintenance activities are needed. Instead, other cloud technologies are affordable, and the scalability and fault-tolerance features can be obtained from the design of the distributed applications themselves.

Another aspect that is strictly connected to our work is the preservation of the metaheuristic nature of GAs, thus allowing the system to be adapted to a wide variety of problems. Even though being related to the EAs in general, the first attempt of generalisation was given from Fazenda et al. [13], who considered the parallelisation of EAs on the Hadoop MapReduce platform in a general purpose form of a library. The work has been further enhanced by Veeramachaneni et al. to produce *FlexGP* [33], which is probably the first large-scale Genetic Programming system that runs in the cloud, implemented over *Amazon EC2* with a socket-based client/server architecture. To the same aim, Ferrucci et al. [9,10,14,34] devised and implemented the *elephant56* framework for parallel GAs development, deployment and execution on the Hadoop MapReduce platform, based on the three models of GAs parallelisation (i.e., the global, grid and island model). They described the design of the framework and how a developer could interact in defining his/her genetic operators or using some provided samples. We addressed this aspect by using software containers that allow defining any environment for GAs execution.

With cloud computing is possible to address almost any problem, by mixing a large variety of technologies. Moreover, some software engineering methodologies are particularly effective in the cloud field, possibly easing the production of parallel GAs (see Section 2). As a first attempt at employing cloud technologies, Merelo Guervós et al. devised *SofEA* [35], a model for Pool-based EAs in the cloud, an evolutionary algorithm mapped to a central *CouchDB* object store. SofEA provides an asynchronous and distributed system for individuals evaluations and genetic operators application. Later, they defined and implemented the *EvoSpace Model* [36], consisting of two components: a repository storing the evolving population and some remote workers, which execute the actual evolutionary process. It is the first work to involve technologies on the *Platform-as-a-Service* (PaaS) and *Software-as-a-Service* (SaaS) level: *Heroku* as PaaS for the population store and *PiCloud* as SaaS for the computing operations. Not only does the work show how EAs can scale on the cloud, but also how the cloud can make EAs effective in a real world environment, speeding up the running time and lowering the costs.

This paper investigates how GAs can effectively take advantage of the cloud to speed up their execution.

4. Background

In this section, we give some background about the involved technologies and communication protocols. The container-based virtualisation and its related most famous utility, i.e., Docker, are presented, respectively, in Sections 4.1 and 4.2. Section 4.3 describes CoreOS, the technology of containers distributed orchestration we employed whereas Section 4.4 illustrates the *Advanced Message Queueing Protocol* (AMQP) we used for communication together with its most famous implementation, i.e., RabbitMQ.

4.1. Container-based virtualisation

The basic idea behind the classic *hypervisor-based virtualisation* is to emulate the underlying physical hardware, creating a new virtual one and installing a fully working *Operating System* (OS) on it. It is the typical model adopted by cloud providers, because of its ability to make hardware shareable and easily maintainable. Even though there are many existent techniques to optimise resources sharing (e.g., the *bare metal virtualisation*), the hypervisor-based virtualisation can be considered as limited in terms of performance. It is true especially when the aim is to execute cloud applications, where several service instances may require to be created and destroyed in seconds to guarantee the reliability and scalability of the entire system.

While with the hypervisor-based virtualisation everything is performed on the hardware level, the *container-based virtualisation* operates at the OS level. It provides a lightweight virtual environment, i.e., the *software container*, that groups and isolates a set of processes and hardware resources from the host and any other container. The main difference with the hypervisor-based virtualisation consists in the fact that all containers share the same kernel of the host system, instead of virtualising it, resulting in a high-performance resource utilisation. For this reason, containers are also much smaller and lightweight compared to an entire virtualised OS [17]. With the isolation, a process inside the container cannot directly see a process or resource outside the container itself, and the network is the only vehicle for communication.

Containers and its features are not such a new technology. Indeed, it was 1979 when it was made possible, for the first time, to create a new root filesystem inside an existing one, using a feature named ‘chroot’. This isolation feature was then evolved into the Linux ‘namespaces’ technology that not only does it offer the isolation of the filesystem, but also of other system resources such as network interfaces. In this way, processes can run in an environment where the resources appear to be dedicated to them.

The term 'process container' was first used around late 2006, then renamed to 'control groups' (abbreviated as 'cgroups') in 2007, as a Linux kernel feature available since v2.6.24. While namespaces isolates processes, cgroups lets the user limit the hardware resources for them. The combination of namespaces and cgroups is the basis of the modern *Linux Containers* (LXC) on which Docker was built on[1] and that Docker simplifies, especially when the aim is to containerise applications.

4.2. Docker

Docker is an open source container orchestration engine that separates applications from the underlying Linux OS. With Docker it is possible to manage software containers, which are intended to contain every component of an application. From the application perspective, there is no difference between an execution on a dedicated machine and inside the container: the application is run in a short time in a full isolated Linux environment and can find others only by using the network. This reduces drastically the activities of installation and maintenance of applications: configuration management methodologies can define the environments and the application can be tested during the process from development to actual production execution, in a CI fashion (see Section 2.1). Docker creates containers from the 'images', i.e., basically read-only templates. Docker also provides an on line registry called 'Docker Hub' where it is possible to push/pull images to/from it. Docker images and registry allow to instantiate containers without repeating installation and build operations. The images can be created through two different operations: 1. by executing operations directly on running containers and saving their state; 2. by executing 'Dockerfiles', a set of instructions which can be maintained in the same way as the source code. Docker is not the only alternative in the field of containers management (e.g., *runC*, and *rkt*), but it is currently the most mature product.

4.3. CoreOS

If Docker orchestrates containers in a single hosting machine, CoreOS can do it on a distributed cluster [16]. *CoreOS* is an open source lightweight OS based on a build of *Chrome OS* by *Google*. It allows building large and scalable deployments on varied infrastructure simple to manage, focusing on security, consistency and reliability. CoreOS provides only minimal functionalities required to execute applications inside Docker containers.[2]

Fig. 2 shows an instance of execution of our application on CoreOS. To manage the cluster, CoreOS exploits a globally distributed key–value store called 'etcd'. Not only does it allow the CoreOS cluster configuration, but also it can be exploited by users as a central point for automatic applications configuration and discovery of other components in the network. The scheduling of containers is managed by a tool called 'fleet' that serves as a cluster-aware init system. It extends on a cluster scale *systemd*, the modern single machine Linux init system. It accepts the requests of containers allocation and schedules assignments to machines in the cluster on an optimisation basis, probing both cluster and applications health.

A new computational resource can join the cluster if a security key is configured. The only requirement is to run CoreOS as OS. Also, CoreOS allows to run an application in a multi-cloud environment [18]. Indeed, even if running on different machines that are geographically distant, CoreOS let them be seen as a single environment on which Docker containers can be deployed and executed. It means that is possible to use at the same time instances from different cloud providers, surpassing the limitations of the number of machines the cloud providers usually impose upon their users.

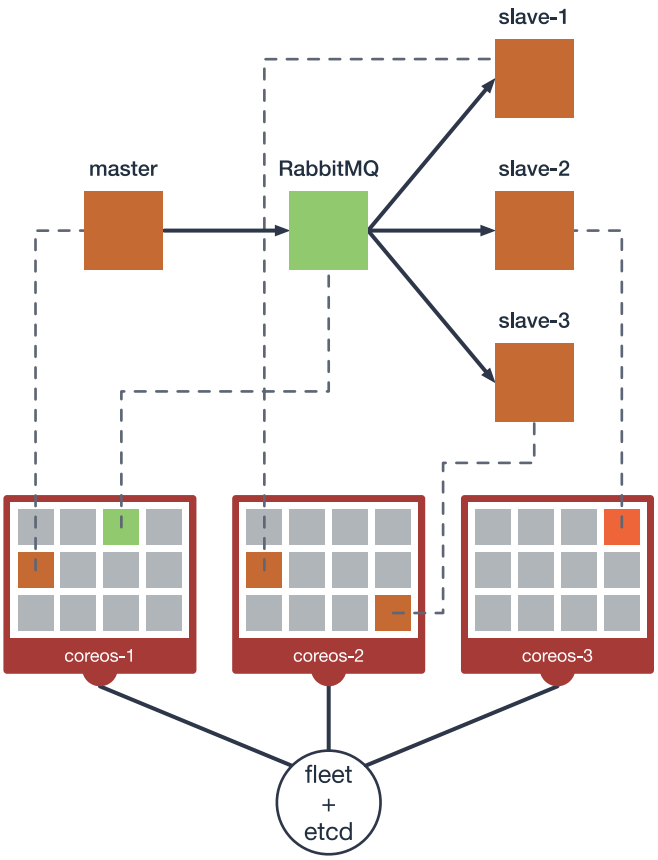

Fig. 2. CoreOS containers distribution.

We preferred to use CoreOS over other alternatives (e.g., *Docker Swarm*, *Kubernetes*, *Mesos*) because they are at the same time lightweight regarding the resource allocation and complete of everything we needed to realise our application. Moreover, it is also available as a cloud instance image on the majority of public cloud providers and thus avoiding the 'lock in' to specific services.

4.4. AMQP and RabbitMQ

RabbitMQ is an open source 'message broker' software that implements the *Advanced Message Queueing Protocol* (AMQP). It is written in *Erlang* language and client libraries are available for the majority of programming languages. It is a component able to accept and forward messages, which can consist of either plain text or blobs of binary data. Message brokers cover each stage of the exchange setup among participants, namely the 'publishers' and 'consumers'. The publishers produce messages and the consumers pick and process them. It is the job of the message broker to ensure that the messages go from a publisher to the right consumer, based on a chosen scheduling policy. The primary recipient of messages is the 'queue', a potentially unlimited buffer of data, which lives inside RabbitMQ. If the publisher and consumers are connected to a queue, they can communicate with each other without actually knowing each other. It makes RabbitMQ a powerful tool for scalable distribution of tasks since it is possible to add and remove participants without breaking the communication.

RabbitMQ has other contestants regarding the AMQP implementation, but no one has, at the same time, a message broker,

[1] Since version 0.9, Docker uses the 'libcontainer' library.

[2] Currently, CoreOS is developing its own container called 'rkt'.

High Availability (HA) capabilities, many client and developer tools available for the majority of programming languages, besides being easily deployable as Docker containers. Furthermore, differently from other communication technologies, RabbitMQ can easily sustain a distributed infrastructure without requiring any other discovery technology.

5. The proposed cloud-based application

In this section, we present the design and implementation of the approach we devised to parallelise GAs in the cloud as a CBA, which we then used as a benchmark for the experimentation described in Section 6. We refer to the development, deployment and execution workflow presented in Section 2, covering each of the expected phases.

We implemented the parallel GA as a distributed algorithm based on containers, following the global parallelisation model (also known as the master/slave model) where a master node executes the GA generations on the whole population except for the fitness evaluation, which is demanded to distributed slave nodes. We named this implementation as *AMQPGA*.

Since we needed other utility components, we structured the application in microservices [37] using separate containers, decoupling functionalities and exploiting existing and more reliable services for communication and report activities. We involved a total of 4 types of services: 1. AMQPGA master, to manage the computation of the whole parallel GA; 2. AMQPGA slave, to compute the fitness evaluation function in parallel for the distributed individuals; 3. RabbitMQ, as the communication protocol and technology; 4. MongoDB, as the recipient of report data for benchmarking and statistics.

In the following, we first describe the design of the architecture of the parallel GA as a CBA in Section 5.1. Once given an overview of the whole application, in Section 5.2 we describe the communication protocol based on AMQP and RabbitMQ. Finally, we describe the algorithms for the master and slaves of AMQPGA in Section 5.3.

5.1. Architecture design

We describe the architecture of the CBA we devised for the parallel GA, including all the specific cloud components and processes indicated in Section 2.

Fig. 3 shows the ensemble of the involved components. The base layer is composed of the cloud infrastructure able to allocate virtual instances of CoreOS, which has been chosen as the cluster manager. With the aim of providing an application as more general and flexible as possible, we can consider indifferently commercial cloud providers (e.g., *DigitalOcean*, *Amazon AWS*, *Windows Azure*) and private cloud environments (e.g., *OpenStack*). It is possible since the only requirement for clusterisation is the availability of the CoreOS image, thus breaking the limits in number and resource usage that single providers may impose on the users. We employed both the main services of CoreOS: fleet as the deployment manager and etcd as the central configuration point for discovery purposes.

As mentioned in Sections 4.2 and 4.3, while CoreOS manages the machines in the cluster and scheduling aspects, Docker manages the download of containers and their execution on the machines assigned by CoreOS. The powerful feature of Docker of executing an entire environment makes possible the implementation of any genetic operator, in any preferred programming language or using any external tool.

The application services of our proposal exploit the underlying interfaces of CoreOS and Docker. One is a running container of RabbitMQ, and the other is the GA cloud implementation, which we named AMQPGA, running in the form of one master and multiple slave containers, communicating through the RabbitMQ service.

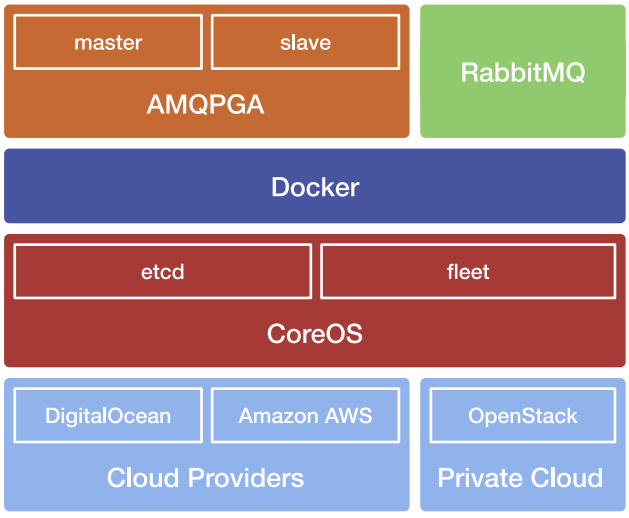

Fig. 3. The involved architecture layers.

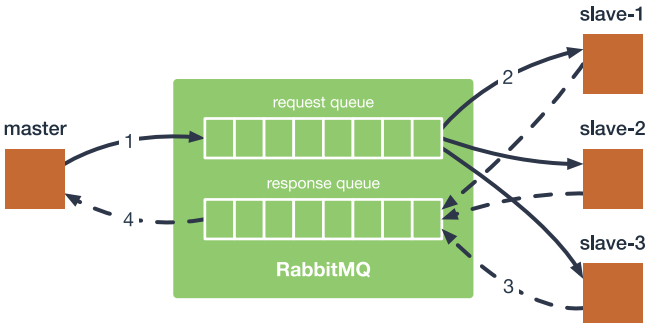

Fig. 4. The AMQPGA algorithm.

Fig. 2 depicts the above situation on the cluster, where CoreOS schedules all the containers to optimise the resources load of the execution.

Regarding the development, deployment and execution workflow described in Section 2, we put our a source code on a *Git* VCS repository. We also implemented a test suite to test the GA components, e.g., the fitness functions. Then, we embedded everything in a software container defining a Dockerfile, that prepares a lightweight environment for the source code execution. We put the resulting Docker image on a public Docker registry, making it available for download. As described in Section 6, we organised the deployment of AMQPGA in the CoreOS cluster in the form of simple scripts that interact with the CoreOS orchestrator, allowing automatic download of AMQPGA image from Docker registry and allocation on cloud resources.

It is worth noting that the indicated services do not require to be embedded in a CBA because they do not strictly require a cloud platform to be executed. Indeed, Docker containers can be executed on any platform having a running Docker engine, allowing independent CI operations.

5.2. AMQP as the communication protocol

We adapted the global model to AMQP model, implemented with a combination of Go workers and a running RabbitMQ service.

The resulting algorithm is an application of the *Remote Procedure Call* (RPC) pattern, depicted in Fig. 4: 1. the AMQPGA master node publishes the messages (i.e., the individuals) on the request queue; 2. RabbitMQ dispatches the individuals to the subscribed

slave nodes, in a round-robin fashion (i.e., assignments are made in equal portions and circular order); 3. the AMQPGA slave nodes process the individuals by computing the fitness function values and publish them on the response queue; 4. the only consumer of the response queue, i.e., the AMQPGA master node, takes back all the individuals and continues the computation until the next generation. Using the message broker as the central point for the computation, we were able to add any number of further slave nodes to the GA, even at runtime, making the application scalable. In terms of discovery, we used the specific name of 'rabbitmq' for the service container executing RabbitMQ. In this way, the master and slave nodes only have to reach that node to be part of the distributed computation.

5.3. Parallel genetic algorithm master and slaves

For the implementation, we chose *Go*, an open-source programming language by Google. In our case, Go was used to simplify the GA processes and build small containerised environments. It is worth noting that it would be possible to switch the Go clients with clients developed with any programming language. The only requirements consist in being able of communicating with RabbitMQ, serialising individuals with the same codification algorithm and respecting the devised communication protocol.

Algorithm 1: The AMQPGA master

```
1  population ← Initialisation(popsize);
2  repeat
3      foreach individual ∈ population do
4          ProduceMessage(individual);
5      population ← ConsumeMessages();
6      selectedCouples ← Selection(population);
7      foreach parent1, parent2 ∈ selectedCouples do
8          child1, child2 ← Crossover(parent1, parent2);
9          offspring ← offspring ∪ {child1} ∪ {child2};
10     foreach individual ∈ offspring do
11         Mutation(individual);
12     population ← SurvivalSelection(population,
       offspring);
13 until termination criterion;
```

Algorithm 1 shows the pseudocode of the AMQPGA master, in charge of managing the communication with the slaves, which compute the fitness evaluation, and the execution of the other genetic operators. It is worth noting that this is just one of the possible implementations for GAs, since we could have also used other genetic operators and customised the order of execution. Moreover, every single operator is generalised since there are several versions, with different parameters each. However, since we mainly aimed at demonstrating the correct operation of the CBA based on software containers and benchmark the parallel GA with the global parallelisation model, the selection and configuration of operators are irrelevant. Let us suppose to have a specific representation for the problem to solve, meaning that is possible to encode the solutions in a data structure, the `Initialisation` function in line 1 produces the initial random population. Until a termination criterion is satisfied, e.g., a specific number of iterations has been reached, the population is evolved through the application of genetic operators lines 2–13, i.e., a generation. At every generation, the master encapsulates each individual in a message that is sent to a queue (`ProduceMessage` function in lines 3–4), ready to be consumed and evaluated from the listening slaves. Then, in line 5 the master collects back all the evaluated individuals through the `ConsumeMessages` function. It is worth noting that the individuals are objects including both the data representation for the solution and the fitness evaluation value. Of course, the fitness value will be filled when computed by the slaves only. Lines 6–12 applies the other genetic operators, i.e., `Selection`, `Crossover`, `Mutation`, and `SurvivalSelection` functions, in the same process of the master, thus being local in respect to the parallel GA.

Algorithm 2: The AMQPGA slave

```
1  while true do
2      individual ← ConsumeMessage();
3      individual ← FitnessEvaluation(individual);
4      ProduceMessage(individual);
```

The AMQPGA slave consists of a straightforward algorithm (Algorithm 2). Until the process is not terminated (line 1), it repeatedly consumes a new message as soon as a new message is ready on the queue on which it is listening (`ConsumeMessage` in line 2). Then, it applies the `FitnessEvaluation` function and fills the fitness value field of the individual in line 3. Each individual is then sent back to the master as a message, `ProduceMessage` in line 4.

The way the individuals are distributed between the slaves is dictated by the configuration of the message queues and communication protocol. Moreover, we set a data exchange with MongoDB to the purpose of collecting experimentation reports.

6. Empirical study design

Our aim was to understand if the proposed approach can be an effective solution to improve the scalability of GAs. Therefore, we had to verify if GAs parallelised using cloud technologies allow us to get a better execution time compared to the sequential version. Moreover, we were interested in quantifying the setup time required to have the infrastructure ready to execute the GAs. Thus, we sought to answer the following research question:

RQ *Is the use of AMQPGA based on a combination of software containers, message queues and cloud orchestration effective for parallel GAs against the sequential execution?*

Considering that the global parallelisation model is parallelised only during the fitness evaluation, to address the **RQ** we considered as a benchmark a dummy fitness evaluation function that does nothing except receiving individuals, sleep for a specified time and return a random fitness value to the master [38]. The choice of this dummy function, together with the variation of the network load (i.e., the chromosome size), was motivated by the fact that it allowed us to assess the GAs scalability considering different problem sizes by just varying the sleep time [5]. Moreover, we tested the actual time required to have the cloud infrastructure ready to execute the parallel GAs. It is worth noting that in global parallelisation model the populations evolve in the same way as the sequential version. For this reason, we do not mention quality results.

Details about the problem and GAs configuration are provided in Section 6.1. The hardware employed to run the experiments is reported in Section 6.2. To understand the effectiveness of the approach, we applied the experimental method described in Section 6.3 and employed several evaluation criteria, namely the execution time, speedup, overhead and setup time, described in Section 6.4. Finally, Section 6.5 analyses some threats to validity that may have affected our experimentation and how we tried to alleviate them.

Table 1
The cloud instances configuration.

Hardware		Software	
Feature	Value	Feature	Value
Architecture	64 bit	CoreOS	1185.5.0
CPUs	1	Docker	1.12.3
RAM	512 MB	RabbitMQ	3.6.6
Storage	20 GB	Go	1.7

6.1. Experiment configuration

To understand how the application behaves, we focused our attention only on the fitness evaluation time, since it is the only parallelisation part of the execution of the global parallelisation model [38]. Nevertheless, we implemented and executed a full GA to reproduce a real scenario in which, besides the execution time, other system resources are consumed, and the GA needs to fit into provided limits (e.g., the memory). The GA has been configured following other studies parameters [36,39]. We initialised a population of 10 000 individuals to let the problem be large enough and splittable to multiple slaves. We ran the GA for 10 generations and varied the chromosome size according to the experimental method (see Section 6.3).

6.2. Hardware

As execution bench for our experiments, we rent several instances from the *DigitalOcean* cloud provider. We composed the cloud clusters employing CoreOS, using 1 instance dedicated to the master, 1 instance for RabbitMQ and varying the size of instances in 1, 2, 4, 8, 16, 32, 64 and 128 for the slaves to test the scalability of the executions. With the aim of maintaining a low budget, we selected only small instances of virtual machines which consisted in 1 core processor, 512 MB of memory and 20 GB of SSD disk for $0.007 h. We made an exception for RabbitMQ node because it requires at least 1 GB of RAM. The configuration of each cloud instance is summarised in Table 1.

6.3. Experimental method

We addressed the **RQ** by comparing the performance of the sequential and parallel GAs with different configurations. We varied the sleep time of the employed dummy fitness function of 0.01 ms, 0.1 ms, 1 ms, 10 ms and 100 ms in order to benchmark different computational times. The network was stressed by varying the individual size (i.e., the chromosome size) in 128, 256, 512, 1024, 2048, 4096, 8192, 16 384, 32 768 and 65 536 genes, where each gene is encoded with 8 bit. We were not able to use a larger chromosome size, due to the memory limits of the sequential execution on a single machine. We executed all the parallel GAs on different cluster configurations characterised by a different number of nodes (see details in Section 6.2). We did not let the master node participate in any parallel fitness computation because we were interested in observing the behaviour of peer communication with RabbitMQ. For each combination of sleep time, chromosome size and cluster configuration, we executed 10 runs. Considering the fitness evaluation as the only actual parallel phase, it allowed us to have a total of 10 generations $\times$ 10 runs $=$ 100 executions to statistically reinforce the observations.

6.4. Evaluation criteria

To compare the performance of the executed experiments, we mostly followed the best practice in reporting the results with parallel GAs, identified by Luque and Alba [2]. We evaluated them both in terms of execution time, speedup and overhead, as detailed in the following. Moreover, we evaluated the setup times of the cloud infrastructure. To cope with the stochastic nature of GAs and hardware executions, we performed some statistical tests.

6.4.1. Execution time

The execution time was measured in milliseconds (ms) using the system clock. As a performance indicator of the whole execution, we compared the execution time achieved by executing all the fitness evaluation phases of sequential and parallel GAs. The partial times were distinguished into computation and overhead times only in a second step when we wanted to quantify the time spent for parallel communication.

6.4.2. Speedup

The speedup is defined as:

$$S = \frac{T_S}{T_P} \tag{1}$$

where T_S is the sequential execution time and T_P the parallel execution time.

We compared the achieved speedup with respect to the ideal speedup, which is equal to the number of the involved parallel nodes and corresponds to the situation when the sequential execution time is perfectly split among multiple nodes. The ideal speedup is rarely achieved in practice due to the presence of overhead, but it is usually taken into account as an upper limit to compare the performance of parallel algorithms [2,40].

6.4.3. Overhead

To understand the reasons that prevent the parallel GAs to have a speedup near to the ideal one, we quantified the overhead for each execution. We considered the time of each execution and distinguished between overhead and computation times. We defined the computation time as:

$$T_C = \frac{T_S}{P} \tag{2}$$

where T_S is the sequential time to compute the fitness evaluation function for the whole population and P the number of parallel slaves. Thus, the overhead time is:

$$T_O = T_P - T_C \tag{3}$$

computable if T_P, i.e., the parallel execution time, is given.

6.4.4. Setup time

One of the points about our approach we took particularly in consideration is its feasibility in a real world context. To this aim, we experimented with the setup time required to have the cloud infrastructure ready to execute the GAs. We discriminated this automated activity into two different times:

- 'creation', the necessary time to acquire the virtual instances from the cloud provider and let all the machines be recognised as part of the CoreOS cluster;
- 'deployment', the time required to pull the GA and RabbitMQ images from the Docker Hub repository, schedule and start the containers on all the machines.

6.4.5. Statistical tests

We executed 10 generations and 10 runs for a total of 100 registered times for each experiment configuration, in order to cope with the inherent randomness of dynamic execution time and reported the average results.

To support all the considerations about the obtained results, we performed the non-parametric inferential statistical test, i.e.,

the *Wilcoxon Signed Rank* test [41], as recommended in the literature [2,20,42]. The Wilcoxon Signed Rank test verifies, as the null hypothesis, if two considered populations have equal distributions. It is particularly useful when no assumptions about the normality of the distributions are possible, as for our case. For all the statistical tests, we accepted a probability of 5 % of committing a Type-I-Error, i.e., the significance level.

Furthermore, we used the *Vargha–Delaney* $\hat{A}_{12}$ test for effect size [43] to characterise the magnitude of difference. The $\hat{A}_{12}$ test is an estimation of the probability the algorithms have against each other in obtaining better results of the considered measure. When two algorithms are compared, and their results are equivalent, then $\hat{A}_{12} = 0.5$. $\hat{A}_{12} > 0.5$ means that, on the average over all the runs, the first algorithm obtains better results than the one with which is compared. The magnitude values can be summarised by 4 nominal values, namely the 'negligible', 'small', 'medium' and 'large'.

6.5. Threats to validity

Threats to *construct validity* concern the relationship between the theory behind the experiments and the observations. In order to alleviate possible threats related to measurement, the GAs execution time was quantified using the system clock, because it represents the speed of a technique to the end-user. We could not make use of any machine independent measure, e.g., the evaluations number, since the sequential and master/slave models behave in the same way.

Threats to *internal validity* concern any confounding factors that could influence our results. A possible threat is related to the randomness due to the use of GAs and variable computational/network load on the nodes at the time of the experiment. Indeed, GAs are intrinsically random, and we mitigated such a threat by executing all the experiments 10 times, with 10 generations each, and presenting the average results [20,42]. Furthermore, the nodes may have been biased by the randomness of system events, and the multiple runs were intended to alleviate these issues as well.

Threats to *external validity* concern the generalisability of our findings outside the scope of our study. An external threat is due to the fact that we benchmarked the parallel GAs on a particular cloud provider (i.e., DigitalOcean) whose machines performance may differ from other providers. For this reason, we carefully separated the times concerning the computation from the setup ones. Moreover, we considered the results of our study as an analysis of the 'execution trends' instead of absolute values, obtained by proportionally varying the configuration parameters of the experiments.

7. Results

In this section, we present the results of our study. The comparison between sequential and parallel GAs, with regarding the execution time, is reported in Section 7.1. The analyses of the speedup and overhead are reported in Sections 7.2 and 7.3, respectively. The setup time is analysed in Section 7.4.

7.1. Execution time

Fig. 5 shows the boxplots of the achieved execution times of the fitness evaluation phase on the whole population, including the time for the communication. Let us recall that we experimented with different combinations of the fitness evaluation time for a single individual (T_f), i.e., the sleep time of the dummy function, the chromosome (c) and cluster sizes (P). Moreover, each generation employed a total of 1000 individuals. We performed 10 runs of 10 generations each, registering a total of 100 observations for each combination.

Table 2
The intervals for which the execution time decreases when increasing the number of parallel nodes.

c	$T_f = 1$		$T_f = 10$		$T_f = 100$	
	P_{min}	P_{max}	P_{min}	P_{max}	P_{min}	P_{max}
128	2	8	2	32	2	128
256	4	8	2	32	2	128
512	4	8	2	32	2	128
1 024	4	8	2	32	2	128
2 048	4	8	2	32	2	128
4 096	4	8	2	32	2	128
8 192	4	8	2	32	2	64
16 384	4	8	2	16	2	64
32 768	4	8	2	16	2	64
65 536	–	–	2	16	2	64

As expected, when increasing the chromosome size, the execution time for the same cluster size and individual fitness time increases proportionally. Focusing on the execution with 1 slave node, for which the communication is not affected by the message broker scheduling policy, we carried out the following statistical tests. We iteratively compared the distributions of consecutive chromosome sizes looking for the first threshold where the *p-value* of the two-tailed Wilcoxon signed-ranks test was less than the level of significance of 0.05 (i.e., accepting the alternative hypothesis of equality). To strengthen the differences, we also employed the Vargha–Delaney test considering as different only the couples having a magnitude level equal to *medium* or *large*. We noticed that the execution time starts to be greater than smaller chromosome sizes only from 2048 genes on, for all the values of the individual fitness times. Even if the passage through a message queues communication system, i.e., the RabbitMQ message broker, adds a certain amount of latency in communication, the chromosome size variation effect is imperceptible if considering that the time is reported in the order of seconds. It can be due to the fact that the network of the employed provider is capable of offering a network speed much higher than what is needed.

As for the scalability on the number of nodes, as easily visible from Fig. 5, the application begins to scale from $T_f = 1$ on. It is clear that there are some minimum and maximum thresholds of the cluster sizes within the approach scales. To support this assumption, we performed a statistical test whose results are shown in Table 2. For each chromosome size and individual fitness time combination, we looked for the first cluster size observations group (P_{min}) whose distribution was significantly greater than the sequential execution one. It was obtained by performing a single-tailed Wilcoxon signed-ranks test setting the level of significance to 0.05. Then, if a minimum threshold was found, we iteratively compared the next couples of consecutive cluster size distributions until the alternative hypothesis by means of the Wilcoxon test was rejected, meaning that the execution time is not significantly decreasing anymore. Thus, we marked that point as the maximum threshold (P_{max}). As shown in Table 2, the approach succeeds in scaling for a few nodes for $T_f = 1$. It is possible to notice that increasing the T_f helps the approach to scale, whereas the chromosome size does not.

On the one hand, because of the communication protocol, the parallel nodes must alternate the phase of receiving, computing of fitness evaluation function and sending of individuals. For this reason, the increment of the chromosome size increases the communication time, even forcing the slaves to be idle for a certain time until the next individuals have been made available from the master.

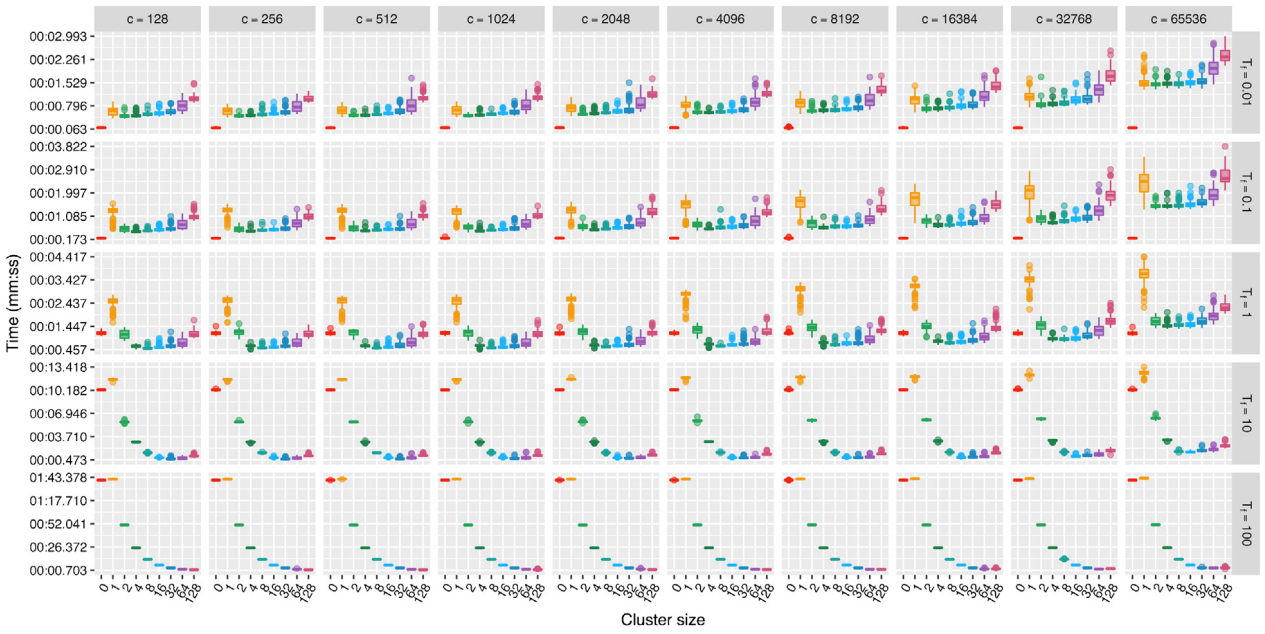

Fig. 5. The execution times achieved by the sequential and parallel GAs, reporting all the combinations for the chromosome size (c), fitness evaluation time (T_f) and cluster size.

On the other hand, the increment of the individual fitness time makes the communication time irrelevant against the computation one thus splitting more linearly the execution times between parallel nodes. Except for the cluster with one node (i.e., size of 1), where the resulting execution time is obviously greater, the clusters with multiple nodes outperform the sequential execution (i.e., the cluster size of 0). It means that the execution time is directly proportional to the individual fitness evaluation time and, as we hoped, inversely proportional to the number of cluster nodes.

The effect of the increment of the number of slave nodes is better discussed in the following, where the speedup is analysed.

7.2. Speedup

The speedup characterises the scalability factor when the number of slave nodes is increasing against the sequential execution. The values are shown in Fig. 6, distinguishing the individual fitness time and the chromosome size. Table 3 reports values on average of the 100 observations for the most positive case of $T_f = 10$ and $T_f = 100$ from the above analysis. As visible from Fig. 5, the GAs begin to scale effectively from $T_f = 1$ on. For $T_f = 10$ and $T_f = 100$, the speedup values tend to the linear speedup according to the thresholds observed in Section 7.1. The observed values suggest that an employment of a T_f having at least a certain complexity, in terms of execution time, is the only requirement that makes the GA based on the master/slave model effectively scalable on multiple nodes, tending to linear scalability.

7.3. Overhead

To further investigate the behaviour of the parallel executions, we analysed the execution time on a more fine-grained scale.

Fig. 7 shows the computation and overhead based on the individual fitness time and chromosome size combinations, where the overhead is intended as the additional time other than the computational one, generally due to communication and message broker (i.e., RabbitMQ) tasks. The stacked bars represent the mean over 100 observations. As we can see from the figure, consistently with the other evaluation criteria, from $T_f = 10$ on the computation time starts to cover the majority of the execution time. Here, it is more evident that the chromosome size only influences the execution time from a certain number of nodes on, a threshold that is shifted accordingly to the individual fitness evaluation time growth.

7.4. Setup time

To understand the feasibility of the approach in a real world context, we analysed the setup time discriminating into the creation and deployment times. As depicted in Fig. 8, both the creation and deployment times are proportional to the target number of nodes but in a light way. In the case of the creation time, it strictly depends on the specific capability of the cloud provider of instantiating new virtual machines. Also, it is comprehensive of the discovery time to let all the nodes be aware of being part of the same cluster. As for the deployment time, it includes the download of the RabbitMQ and AMQPGA images on all the nodes and the actual scheduling of the containers. The considered times are proportional to different influencing factor that can vary based on the context but, in a general way, they can give an optimistic measure of the setup time that takes only 5 minutes to have the infrastructure ready to start the GAs. Even if with the cloud any consumed time has its cost, this setup time is irrelevant if the GA is run for many generations and also it is not required to be repeated for any other GAs executed on the same cluster.

8. Discussion

In this section, we use the results from the previous section as a starting point for a comparative discussion between AMQPGA and other state-of-the-art approaches. We did not include approaches on multi-core (i.e., CPUs) and many-core (i.e., GPUs) computation since they are based on a fixed number of parallel workers and cannot scale on several nodes [3], as distributed systems do. In particular, we selected two most recent open source projects that we believe are the closest to ours.

DEAP[3] (Distributed Evolutionary Algorithms in Python) is a framework that allows writing EAs, thus including GAs, in Python

[3] https://github.com/DEAP/deap.

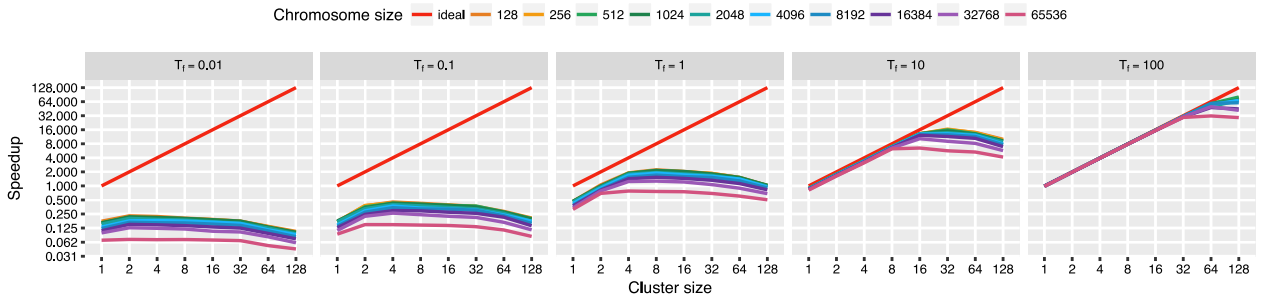

Fig. 6. The speedup trend.

Table 3
The speedup values for $T_f = 10$, $T_f = 100$ and each chromosome size combination.

c	$T_f = 10$								$T_f = 100$							
	1	2	4	8	16	32	64	128	1	2	4	8	16	32	64	128
128	0.877	1.779	3.455	6.957	13.138	16.038	14.058	9.969	0.988	1.975	3.946	7.891	15.628	30.486	56.681	74.939
256	0.877	1.775	3.465	6.958	13.140	16.504	13.831	9.678	0.989	1.975	3.951	7.895	15.585	30.749	52.914	78.033
512	0.879	1.774	3.451	6.967	13.047	15.854	13.215	9.227	0.988	1.977	3.949	7.886	15.616	30.815	57.994	80.539
1 024	0.877	1.770	3.463	6.934	13.146	14.994	12.931	9.210	0.988	1.976	3.949	7.885	15.614	30.645	57.135	69.973
2 048	0.875	1.765	3.434	6.941	13.118	13.807	12.509	8.741	0.989	1.978	3.951	7.888	15.589	30.321	50.488	67.842
4 096	0.861	1.731	3.398	6.777	12.882	13.096	12.135	7.943	0.988	1.973	3.945	7.881	15.553	30.636	55.738	68.073
8 192	0.853	1.709	3.335	6.689	12.528	12.467	11.463	7.642	0.986	1.972	3.943	7.868	15.500	30.403	58.775	60.880
16 384	0.849	1.694	3.295	6.642	12.171	11.476	10.468	6.957	0.985	1.973	3.940	7.868	15.453	30.121	47.978	44.765
32 768	0.832	1.653	3.216	6.501	10.296	9.058	8.170	5.720	0.983	1.971	3.927	7.812	15.382	30.212	49.961	41.780
65 536	0.814	1.628	3.136	6.251	6.484	5.666	5.325	4.158	0.979	1.966	3.919	7.825	15.349	29.406	31.672	29.192

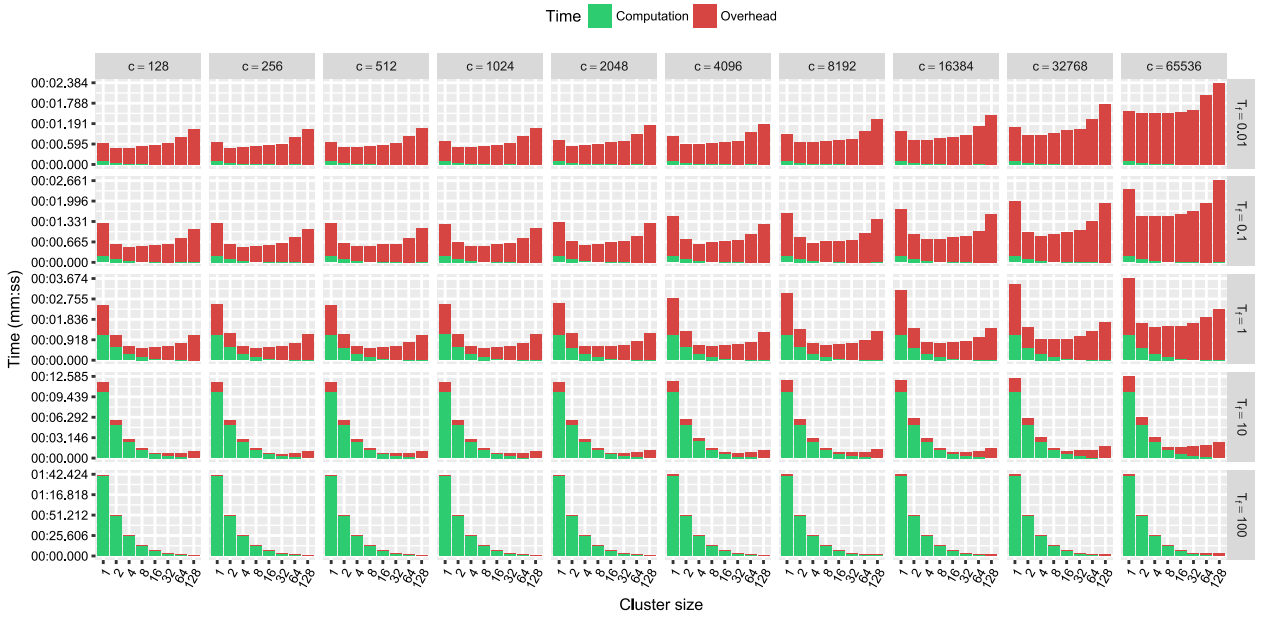

Fig. 7. The computation and overhead times achieved by the sequential and parallel GAs.

[44,45]. It employs *SCOOP*[4] (Scalable COncurrent Operations in Python), a Python module to define distributed tasks for concurrent parallel programming on various environments, including cloud computing, based on *ZeroMQ* as message passing protocol. Once defined the representation encoding and genetic operators, the end user can define the execution flow of the GA. The 'map' function can be then used to apply a single operator, e.g., the fitness evaluation function, to a collection of multiple individuals. Usually, the map function is sequentially executed on the running process. However, DEAP can transform the map function in a parallel or distributed one making use of multi-threading or SCOOP, relatively.

elephant56[5] is a framework to develop GAs to be run on an *Apache Hadoop MapReduce* cluster [9,10,14]. Hadoop is currently based on *Yet Another Resource Negotiator* (YARN), a platform that comprehends also other distributed Apache products [46]. The MapReduce paradigm is expressed in terms of two functions: the 'map' is responsible for handling the parallelisation while the 'reduce' collects and merge results. elephant56 completely hides the parallelisation and MapReduce matters to the end user, who only needs to define and assemble genetic operators for the GA. Then, s/he has to decide which parallel model for GAs to use, e.g., the

4 https://github.com/soravux/scoop/.

5 https://github.com/pasqualesalza/elephant56.

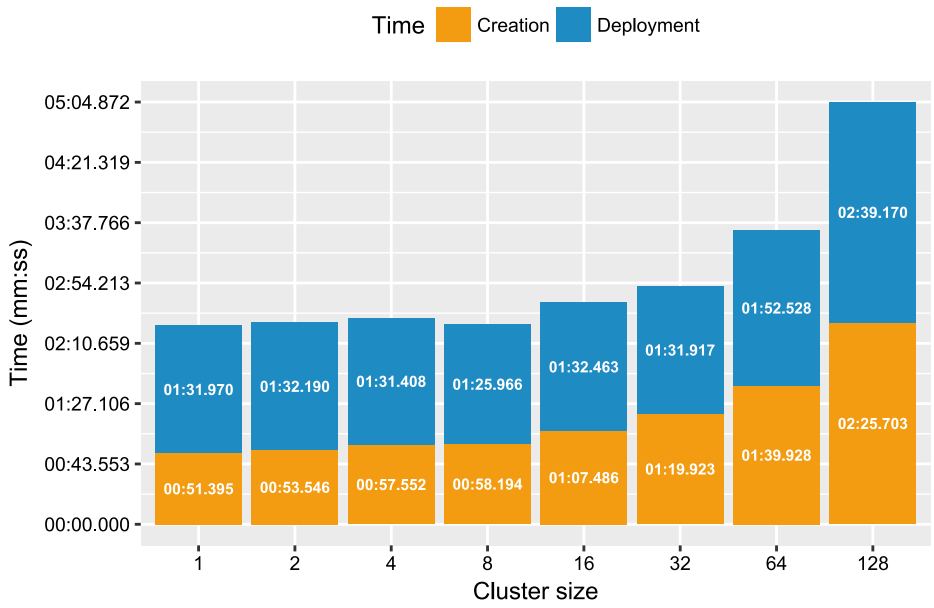

Fig. 8. The setup times achieved by requesting different cluster sizes.

global, grid, and island, which will be automatically translated into a proper MapReduce model. The end user can define his/her GA using the implementations of individual encoding and genetic operators already provided by the framework or implement some new as Java classes. Once packed in a *Java ARchive* (JAR) file, Hadoop is in charge of the distributed computation in the form of a job. Hadoop distributes the jobs on a cluster of nodes by using *Java Virtual Machines* (JVMs) for the computation and HDFS for the reliable storage and passage of data.

In the following, we discuss some aspects in detail.

8.1. Genetic algorithms engineering

We mostly refer to what we presented in Section 2, where we described the activity of applying modern software engineering methodologies to develop, deploy and execute GAs as CBAs.

8.1.1. Invoking external components

As presented above, both DEAP and elephant56 allow defining the encoding of solutions and genetic operators, for both single- and multi-objective evaluations. They also already provide some implementations of common components on the shelf. Let us focus on one of the elements we think make our approach more flexible as it is by design: the possibility of invoking external components as genetic operators. Indeed, many implementations could be available only in a particular programming language, e.g., a suite for machine learning. In AMQPGA everything is packed in a software container, which is a collection of the software to execute but also of the environment configuration to run it. Using Dockerfiles, it is possible to define the needed components to run the GA itself, e.g., Go for AMQPGA master and slave executables, together with other custom components. For instance, one could install a JVM to run a Java application, or need a Python environment to run some scripts. Then, the external components can be invoked by means of simple shell calls to the underlying environment, as the component were a black box and for which the GA only needs to interact with it in terms of input and output. In the case of AMQPGA, the customisation of the environment is included by design.

DEAP basically consists of Python code, and it is possible to invoke shell commands. What is not already provided is an easy customisation of the environment. However, DEAP can be easily included in containers even if this behaviour is not expected by design.

With elephant56 the things are a bit more complicated. Hadoop executes the distributed jobs inside JVMs, running on computational nodes of the cluster. It is possible to temporary go outside the JVM to invoke external commands. However, the customisation of the environments requires that the end user would prepare every node installing all the required components. Besides being a time-consuming operation, it can also be not feasible in some cases. Indeed, it is required to have administrators permissions to have access to Hadoop nodes and install something on them. The administrator of the Hadoop cluster might not allow that since any modification could potentially compromise the health of the entire cluster.

8.1.2. Continuous integration

As for the testing aspects, we refer mostly on local testing, allowing typical CI activities. It consists in simulating sequential and parallel execution on a single machine. We skip a discussion about the testing of the frameworks themselves since their software units can be easily tested with traditional methodologies. Also, we exclude from the discussion the test for encoding and genetic operators, which can be obtained similarly. Thus, we refer to the GA application built upon them.

Being AMQPGA based on software containers, it makes no difference if the containers are run on a cloud cluster or a single machine. Moreover, containers are a lightweight virtualisation, thus requiring very low overhead on resources to be executed. Also, RabbitMQ and MongoDB can be executed on a single machine, therefore allowing a full simulation of the whole system.

DEAP includes testing activities both for sequential executions and parallel ones. The parallel execution can be obtained by making use of the Python ‘multiprocessing’ module that DEAP fully supports. Unfortunately, the test of the distributed version using SCOOP is excluded by design. However, the components could be encapsulated in software containers and the run in the same machine.

elephant56 allow testing in local, but it is easily achievable with one node only. The first option, i.e., ‘standalone’ mode, consists of using Hadoop entirely in local, avoiding the use of HDFS. It allows a very lightweight execution since all parallel functions are run using

a single JVM. However, sacrificing the test of HDFS could be not desirable because it is one of the key points of the Hadoop jobs. Moreover, it is not possible to use multiple reduce executions that is fundamental to test grid and island models. Instead, the second option that is called 'pseudo-distributed' simulates all the required Hadoop daemons, each one running on an independent JVM, thus resulting in very heavy executions possibly not runnable with CI.

8.1.3. Continuous deployment

As for the CD, we consider and discuss the possibility of using a cluster of multiple machines and extending the size of it when and if needed.

AMQPGA is specifically devised to run in the cloud using a container orchestration platform, e.g., CoreOS. The orchestration platform is directly in charge of scheduling containers on the connected resources. Moreover, as described in Section 4.3, a new node can join the cluster by simply installing CoreOS on it. As shown in Section 7.4, the installation of a cloud cluster of 128 nodes takes around 5 min. Furthermore, being AMQPGA based on RabbitMQ, a new node can join a running computation even during a GA is running, in a plug and play fashion. The AMQPGA nodes use the discovery feature of CoreOS to reach the RabbitMQ queue, and then they are ready to divide the computation with the others.

DEAP uses SCOOP for distributed tasks. SCOOP is based on *Secure SHell* (SSH) to connect to remote hosts, thus using SSH keys as a join condition to the same cluster. Even if SSH protocol might result troublesome, DEAP can be potentially easily deployed. It only expects the resulting machine to have installed a Python environment, which is a very common configuration offered by commercial cloud providers. However, the master node needs to know a priori the list of the available resources with the relative addresses, meaning that is not possible to join the distributed execution while a GA is running.

elephant56, being a Hadoop application, is strictly related to the underlying platform. Maintaining and extending a Hadoop cluster means having specific expertise with YARN. Indeed, every single node needs to be set up by installing the OS first, the YARN platform and enabling proper SSH connection with the rest of the nodes. Therefore, it can result in an effort-prone activity.

8.2. Performance discussion

Here we further discuss the results of the empirical study we conducted, by comparing with the way DEAP and elephant56 operate.

8.2.1. Execution time

In Sections 7.1 and 7.3 we analysed the execution time and overhead when running several GA experiments. We varied the chromosome size, i.e., the quantity of data required to represent a single individual, and the fitness evaluation time, i.e., the required time to evaluate a single individual. As also demonstrated by related work [11,12,14,30,31], distributed GA are expected to be particularly effective with intensive computation, when the fitness evaluation time overcome the overhead. Intuitively, the chromosome size seems to be the main factor for overhead. It is strictly related to the activity of communication, consisting of the data that should be moved from one node to another. However, also the specific characteristics of the communication protocol can largely impact performance. In the case of AMQPGA, the presence of a message broker causes every message to pass through a middle point, be stored inside a queue, i.e., a data structure, before being sent to the final destination. On the other hand, it guarantees full reliability in message exchange, and there is no risk if a message gets lost during transmission. Moreover, as described above, the nodes can join the computation in a plug and play way at any moment.

In the case of DEAP, the presence of SCOOP as distributed manager lightens a bit the communication. It depends on the fact that SCOOP uses ZeroMQ, which implements a message queue protocol for asynchronous communication but without using a message broker. However, a master node needs to be fully in charge of resource scheduling reducing the reliability of the whole cluster.

As for elephant56, the data passes through HDFS. On the one hand, data is distributed over the nodes in the Hadoop cluster with a redundancy protocol that guarantees the reliability of data. On the other hand, as confirmed by the literature [14], HDFS is an important bottleneck for communication. The application JAR files have also to be transmitted all over the network when the job starts, adding further overhead.

9. Conclusions and future work

In this paper, we distributed *Genetic Algorithms* (GAs) based on the master/slave model with technologies specifically devised for the cloud, i.e., the software containers, cloud orchestration and message queues. We presented a novel implementation, called AMQPGA, that exploits message queues to schedule parallel GAs tasks. We also devised a conceptual workflow for development, deployment and execution activities of distributed GAs as *Cloud-Based Applications* (CBAs), exploiting modern software engineering methodologies and tools. Then, we empirically assessed the effectiveness of the approach in terms of execution time, speedup, overhead, using a dummy fitness function as a benchmark problem. Finally, we compared AMQPGA with state-of-the-art approaches, highlighting the pros and cons of using an architecture based on containers.

We succeeded in accelerating the execution time of our GA application up to a total number of 128 slave nodes. From the results, it emerged that there is a dependency between computation load and communication cost. We observed that the execution time is directly proportional to the individual fitness evaluation time and inversely to the number of cluster nodes. There is an inferior limit for the evaluation time for the fitness function that makes the parallelisation effective. Also, there is also a superior limit regarding the chromosome size that, together with the population size, determines the network load and thus influences the final execution time. Moreover, we observed that the setup time can be quantified to a few minutes even if the request is of many nodes (e.g., 128). It is worth noting that this time is related to a completely automatised activity, which does not require the human presence as in the case of other methodologies, e.g., Hadoop [10].

The performance and setup times place the cloud positively between other employed technologies for GAs parallelisation, e.g., multi-core systems, GPUs [3,4] and Hadoop MapReduce. Cloud orchestration and software containers surpass limitations of existing approaches for parallel GAs distribution, offering exclusive features such as environment configuration as part of the development and plug and play join to the computation. Also, it is clear that cloud orchestration for parallel GAs can be considered affordable in an economical way, against other technologies strictly related to the hardware physically owned.

One avenue for future work is to evaluate other models of parallel GAs such as the cellular and island model [2]. The empirical study should be replicated with other fitness functions, and we want to put into operation our structure by solving real world optimisation problems, such as Test Suite generation [12,31] or machine learning problems. To make the approach more flexible and easy to use, we also plan to abstract the concepts further and propose it in the form of a framework. In this way, the developer would have to deal exclusively with the activity of problem encoding. As for the cloud aspects, we want to measure other metrics

such as energy efficiency and security of the algorithms [47]. We aim at exploiting and investigating more the features of multi-cloud [18], enhancing our architecture with an Intention Workflow Model [15,48] to reach full self-management of GAs execution in the cloud, and considering the cost-effect factor of cloud spot instances [49].

References

[1] M. Harman, The current state and future of search based software engineering, in: 2007 Future of Software Engineering, IEEE Computer Society, 2007, pp. 342–357.
[2] G. Luque, E. Alba, Parallel Genetic Algorithms: Theory and Real World Applications, in: Studies in Computational Intelligence, no. 367, Springer, 2011.
[3] L. Zheng, Y. Lu, M. Guo, S. Guo, C.-Z. Xu, Architecture-Based design and optimization of genetic algorithms on multi- and many-core systems, Future Gener. Comput. Syst. 38 (2014) 75–91.
[4] S. Yoo, M. Harman, S. Ur, GPGPU test suite minimisation: search based software engineering performance improvement using graphics cards, Empir. Softw. Eng. 18 (3) (2013) 550–593.
[5] M. Ivanovic, V. Simic, B. Stojanovic, A. Kaplarevic-Malisic, B. Marovic, Elastic grid resource provisioning with wobingo: a parallel framework for genetic algorithm based optimization, Future Gener. Comput. Syst. 42 (2015) 44–54.
[6] D. Lim, Y.-S. Ong, Y. Jin, B. Sendhoff, B.-S. Lee, Efficient hierarchical parallel genetic algorithms using grid computing, Future Gener. Comput. Syst. 23 (4) (2007) 658–670.
[7] P. Salza, F. Ferrucci, F. Sarro, Develop, deploy and execute parallel genetic algorithms in the cloud, in: Evolutionary Computation Software Systems Workshop at GECCO, EvoSoft, 2016, pp. 121–122.
[8] I. Sadooghi, J.H. Martin, T. Li, K. Brandstatter, K. Maheshwari, T.P.P. de Lacerda Ruivo, G. Garzoglio, S. Timm, Y. Zhao, I. Raicu, Understanding the performance and potential of cloud computing for scientific applications, IEEE Trans. Cloud Comput. 5 (2) (2017) 358–371.
[9] F. Ferrucci, M.-T. Kechadi, P. Salza, F. Sarro, A framework for genetic algorithms based on hadoop, Computing research repository (CoRR) abs/1312.0086.
[10] F. Ferrucci, P. Salza, M.-T. Kechadi, F. Sarro, A parallel genetic algorithms framework based on hadoop mapreduce, in: ACM/SIGAPP Symposium on Applied Computing, SAC, 2015, pp. 1664–1667.
[11] A. Verma, X. Llorà, D.E. Goldberg, R.H. Campbell, Scaling genetic algorithms using mapreduce, in: International Conference on Intelligent Systems Design and Applications, ISDA, 2009, pp. 13–18.
[12] L. Di Geronimo, F. Ferrucci, A. Murolo, F. Sarro, A parallel genetic algorithm based on hadoop mapreduce for the automatic generation of junit test suites, in: IEEE International Conference on Software Testing, Verification and Validation, ICST, 2012, pp. 785–793.
[13] P. Fazenda, J. McDermott, U.-M. O'Reilly, A library to run evolutionary algorithms in the cloud using mapreduce, in: European Conference on Applications of Evolutionary Computation, EvoApplications, 2012, pp. 416–425.
[14] F. Ferrucci, P. Salza, F. Sarro, Using hadoop mapreduce for parallel genetic algorithms: a comparison of the global, grid and island models, Evolutionary Computation.
[15] T. Baker, M. Mackay, M. Randles, A. Taleb-Bendiab, Intention-Oriented programming support for runtime adaptive autonomic cloud-based applications, Comput. Electr. Eng. 39 (7) (2013) 2400–2412.
[16] D. Weerasiri, M.C. Barukh, B. Benatallah, Q.Z. Sheng, R. Ranjan, A taxonomy and survey of cloud resource orchestration techniques, ACM Comput. Surv. 50 (2) (2017) 1–41.
[17] Z. Kozhirbayev, R.O. Sinnott, A performance comparison of container-based technologies for the cloud, Future Gener. Comput. Syst. 68 (2017) 175–182.
[18] N. Ferry, A. Rossini, F. Chauvel, B. Morin, A. Solberg, Towards model-driven provisioning, deployment, monitoring, and adaptation of multi-cloud systems, in: IEEE International Conference on Cloud Computing, Cloud, 2014, pp. 887–894.
[19] M. Harman, S.A. Mansouri, Y. Zhang, Search-Based software engineering: trends techniques and applications, ACM Comput. Surv. 45 (1) (2012) 1–61.
[20] M. Harman, P. McMinn, J.T. de Souza, S. Yoo, Search Based Software Engineering: Techniques, Taxonomy, Tutorial, in: Empirical Software Engineering and Verification, Vol. 7007, Springer, Berlin Heidelberg, 2012, pp. 1–59.
[21] M. Harman, B.F. Jones, Search-Based software engineering, Inf. Softw. Technol. 43 (2001) 833–839.
[22] X. Yu, M. Gen, Introduction to Evolutionary Algorithms, Decision Engineering, Springer London, London, 2010.
[23] elephant56, A genetic algorithms framework for hadoop mapreduce. https://github.com/pasqualesalza/elephant56.
[24] jMetal: a framework for multi-objective optimization with metaheuristics. https://github.com/jMetal/jMetal.
[25] Jenetics, Java genetic algorithm library. http://jenetics.io.
[26] DEAP, Distributed evolutionary algorithms in Python. https://github.com/DEAP/deap.
[27] M. Fowler, VersionControlTools. https://martinfowler.com/bliki/VersionControlTools.html (February 2010).
[28] M. Fowler, Continuous integration. https://www.martinfowler.com/articles/continuousIntegration.html (January 2006).
[29] Agile Alliance, Continuous Deployment, 2018. https://www.agilealliance.org/glossary/continuous-deployment.
[30] C. Jin, C. Vecchiola, R. Buyya, MRPGA: an extension of mapreduce for parallelizing genetic algorithms, in: IEEE International Conference on E-Science (e-Science), 2008, pp. 214–221.
[31] S. Di Martino, F. Ferrucci, V. Maggio, F. Sarro, Towards migrating genetic algorithms for test data generation to the cloud, in: Software Testing in the Cloud: Perspectives on an Emerging Discipline, IGI Global, 2013, pp. 113–135.
[32] I.A.T. Hashem, N.B. Anuar, A. Gani, I. Yaqoob, F. Xia, S.U. Khan, MapReduce: review and open challenges, Scientometrics 109 (1) (2016) 389–422.
[33] K. Veeramachaneni, I. Arnaldo, O. Derby, U.-M. O'Reilly, FlexGP: cloud-based ensemble learning with genetic programming for large regression problems, J. Grid Comput. 13 (3) (2015) 391–407.
[34] P. Salza, F. Ferrucci, F. Sarro, Elephant56: design and implementation of a parallel genetic algorithms framework on hadoop mapreduce, in: Genetic and Evolutionary Computation Conference, GECCO, 2016, pp. 1315–1322.
[35] J.J. Merel Guervós, A.M. Mor. García, C.M. Fernandes, A.I. Esparcia-Alcázar, SofEA, a pool-based framework for evolutionary algorithms using couchDB, in: Genetic and Evolutionary Computation Conference, GECCO, 2012, pp. 109–116.
[36] M. García-Valdez, L. Trujillo, J.J. Merel Guervós, F. Fernandez de Vega, G. Olague, The evospace model for pool-based evolutionary algorithms, J. Grid Comput. 13 (3) (2015) 329–349.
[37] J. Lewis, M. Fowler, Microservices, a definition of this new architectural term. https://martinfowler.com/articles/microservices.html (March 2014).
[38] E. Cantú-Paz, D.E. Goldberg, On the scalability of parallel genetic algorithms, Evol. Comput. 7 (4) (1999) 429–449.
[39] E. Alba, A.J. Nebro, J.M. Troya, Heterogeneous computing and parallel genetic algorithms, J. Parallel Distrib. Comput. 62 (9) (2002) 1362–1385.
[40] E. Alba, J.M. Troya, Analyzing synchronous and asynchronous parallel distributed genetic algorithms, Future Gener. Comput. Syst. 17 (4) (2001) 451–465.
[41] W.J. Conover, Practical Nonparametric Statistics, third ed., John Wiley & Sons, 1999.
[42] A. Arcuri, L. Briand, A practical guide for using statistical tests to assess randomized algorithms in software engineering, in: IEEE/ACM International Conference on Software Engineering, ICSE, 2011, pp. 1–10.
[43] A. Vargha, H.D. Delaney, A critique and improvement of the "cl" common language effect size statistics of mcgraw and wong, J. Educ. Behav. Stat. 25 (2) (2000) 101–132.
[44] D. Rainville, F.-A. Fortin, M.-A. Gardner, M. Parizeau, C. Gagné, DEAP: a python framework for evolutionary algorithms, in: Genetic and Evolutionary Computation Conference (GECCO), ACM, 2012, pp. 85–92.
[45] F.-A. Fortin, F.-M.D. Rainville, M.-A. Gardner, M. Parizeau, C. Gagné, DEAP: evolutionary algorithms made easy, J. Mach. Learn. Res. 13 (2012) 2171–2175.
[46] T. White, Hadoop: The Definitive Guide, O'Reilly Media, Inc., 2012.
[47] B. Aldawsari, T. Baker, D. England, Trusted energy-efficient cloud-based services brokerage platform, Int. J. Intell. Comput. Res. 6 (4) (2015) 630–639.
[48] T. Baker, O.F. Rana, R. Calinescu, R. Tolosana-Calasanz, J.A. Bañares, Towards autonomic cloud services engineering via intention workflow model, in: International Conference on Economics of Grids, Clouds, Systems, and Services, GECON, 2013, pp. 212–227.
[49] Z. Li, H. Zhang, L. O'Brien, S. Jiang, Y. Zhou, M. Kihl, R. Ranjan, Spot pricing in the cloud ecosystem: a comparative investigation, J. Syst. Softw. 114 (2016) 1–19.

Pasquale Salza is a Postdoctoral Research Assistant at USI Università della Svizzera italiana, Switzerland. He received his Ph.D. degree in Computer Science from the University of Salerno, Italy, in 2017. His research interests are mainly focused on cloud computing, search-based software engineering, evolutionary computation and mobile computing, with the aim of efficiently joining solutions and approaches to improve Information Technology systems and supply software solutions of better quality, cost-effectively.

Filomena Ferrucci is professor of software engineering and software project management at University of Salerno, Italy. Her main research interests include software metrics, effort estimation, search-based software engineering, empirical software engineering, and human–computer interaction. She has been program co-chair of the International Summer School on Software Engineering.