Develop, Deploy and Execute Parallel Genetic Algorithms in the Cloud

Pasquale Salza[1], Filomena Ferrucci[1], Federica Sarro[2]
[1]University of Salerno, Italy – [2]University College London, United Kingdom
psalza@unisa.it, fferrucci@unisa.it, f.sarro@ucl.ac.uk

ABSTRACT

Making Genetic Algorithms (GAs) distributed in an on-demand fashion involves different phases from resources allocation to actual deployment and execution. We propose a cloud architecture with a conceptual workflow able to cover each GAs distribution phase.

CCS Concepts

•Computing methodologies → Genetic algorithms; *Massively parallel algorithms;* **•Networks → Cloud computing;**

Keywords

Parallel Genetic Algorithms; Cloud Computing; DevOps; Container Virtualisation

1. INTRODUCTION

Parallelism may allow reducing the computational time of Genetic Algorithms (GAs) addressing the scalability issue posed by real-world problems. Nevertheless, suitable technologies are needed to ensure both reliability and good performance, considering that GAs require continuous data exchange. Previous proposals for distributed GAs exploited well known technologies such as Hadoop MapReduce [3, 5] and AppEngine MapReduce [2], however these technologies exchange data through a distributed file system or datastore, which may slow down the entire execution [1]. Other implementations consist of ad hoc solutions that reduce the communication to the minimum [4]. All these solutions are not designed to take advantage from the most appealing features of cloud computing, namely the on-demand resources allocation. Cloud also eases the development and deployment processes if used as the environment for the DevOps ('development' and 'operations') methodology. This approach suits well those cases in which the deployment of an application in a distributed and multi-component system is needed . Moreover, it drastically reduces the activities of installation and maintenance: the environments can be defined by configuration management methodologies and the application can be tested during the process from the development to the actual execution. From a developer perspective, which aims to distribute a parallel GA, it is crucial that the execution environment allows him/her to define the genetic operators in any programming language.

In this work we design an architecture for parallel GAs (see Section 2) based on two technologies specifically devised for the cloud (i.e., Docker and CoreOS) that allow us to use isolated application environments (i.e., containers). This architecture is part of a conceptual workflow (see Section 3) devised to support both developers and users from the development phase to the deployment and execution of distributed GAs.

ACM ISBN 978-1-4503-4323-7/16/07.
DOI: http://dx.doi.org/10.1145/2908961.2909024

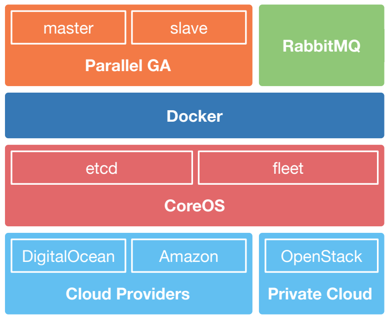

Figure 1: The architecture layers.

2. THE PROPOSED ARCHITECTURE

Figure 1 shows the architecture of the system. The bottom layer consists of the cloud infrastructure. On top of it there is the cluster manager `CoreOS`, an open source lightweight operating system that allows us to easily build large and scalable deployments on different infrastructures, focusing on security, consistency and reliability. We exploit two main CoreOS components: `fleet` as deployment manager and `etcd` as central configuration point, which allows distributed applications to see each other.

CoreOS provides only the minimal functionalities required by `Docker` to execute the applications at the above level. Docker is an open source container orchestration engine that separates applications from the underlying operating system (in our case CoreOS). It allows us to instantiate 'application containers', which are intended to contain all the components of an application. For the application, there is no differ-

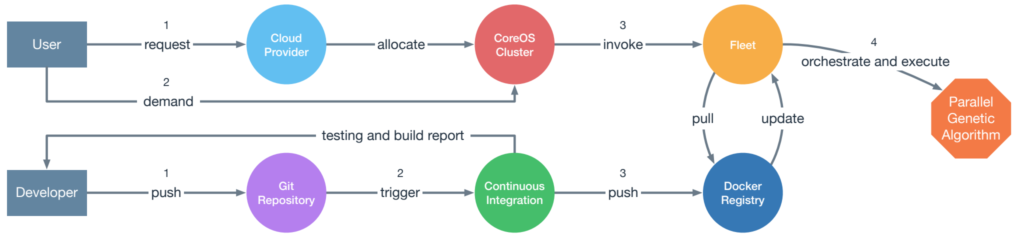

Figure 2: Genetic Algorithms development, deployment and execution workflow.

ence between an execution on a dedicated machine or inside the container, because the application is quickly executed in a full isolated environment and it can discover others containers by relying on the communication network. If Docker is able to orchestrate containers in a single hosting machine, CoreOS enables us to do it on a distributed cluster. Docker creates containers out of `images` (i.e., read-only templates) and provides an on-line registry called `Docker Hub` where to push/pull these images. Images and registry are fundamental components within the deployment process, since they permit instantiating containers without repeating installation and build operations. The powerful feature of Docker to execute an entire environment (i.e., virtual Linux instances) allows the GAs developers to implement the genetic operators in their preferred programming language.

The top layer of our architecture contains the two main components used to define GAs by exploiting the underneath interface provided by CoreOS and Docker. The first component is a running container of `RabbitMQ`, an open source 'message broker' that implements the Advanced Message Queueing Protocol (AMQP). Its role is to ensure that data (i.e., messages) go from a publisher, who produces messages, to consumers, who process them. The main recipient of messages is the 'queue', a potentially unlimited buffer of data, which resides inside RabbitMQ. If publisher and consumers are connected to the same queue, they can communicate without actually know each other. This makes RabbitMQ a powerful tool for scalable distribution of tasks.

The second component is formed by multiple containers responsible of both communication and genetic operators implementations. As an example let us consider the global parallelisation model [2] where the GAs architecture is composed by a master node executing the genetic operators and some distributed slave nodes to which the fitness evaluation is demanded. Once the fitness values have been computed, the individuals go back to the master node where the genetic operators are applied to produce the next offspring. In our proposal this data (i.e., the individuals) would be exchanged through the RabbitMQ service rather than using a distributed file system as proposed in previous work. Using the message broker as the master node for the computation enables to add any further slave nodes to the GA, even at running time, making the system scalable.

3. THE CONCEPTUAL WORKFLOW

Figure 2 shows a conceptual workflow that can be applied thought our architecture. It involves two actors: a developer deploying a distributed GA and a user executing it. In particular, ① the developer pushes its code to its public or private Git repository, together with a Docker file defining the environment; ② after the push, the Git repository automatically triggers (hook mechanism) a continuous integration service that executes both the integration testing of the new source code and the Docker image building; ③ if both testing and building succeed, a report is sent to the developer and the Docker image is pushed in a Docker registry.

The Docker image of the GA developed is now ready to be deployed to a production cluster and executed by the user as follows: ① the user submits a request to a cloud provider and a cluster with the number of CoreOS nodes s/he requested is allocated; ② the user demands CoreOS to execute the GA with a certain configuration; ③ CoreOS invokes fleet which pulls the Docker image of the GA implementation and any other useful service images (RabbitMQ in our case) from the Docker registry, if there is a newer version available; ④ fleet is ready to orchestrate containers and start the execution of the distributed GA.

4. CONCLUSIONS

This work proposes a novel architecture to develop, deploy and execute parallel Genetic Algorithms in a cloud environment. As future work we plan to empirically evaluate the approach in order to assess its scalability.

5. REFERENCES

[1] L. Di Geronimo, F. Ferrucci, A. Murolo, and F. Sarro. A Parallel Genetic Algorithm Based on Hadoop MapReduce for the Automatic Generation of JUnit Test Suites. In *IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pages 785–793. IEEE, 2012.

[2] S. Di Martino, F. Ferrucci, V. Maggio, and F. Sarro. Towards Migrating Genetic Algorithms for Test Data Generation to the Cloud. In *Software Testing in the Cloud: Perspectives on an Emerging Discipline*, pages 113–135. IGI Global, 2013.

[3] F. Ferrucci, P. Salza, M.-T. Kechadi, and F. Sarro. A Parallel Genetic Algorithms Framework Based on Hadoop MapReduce. In *ACM/SIGAPP Symposium on Applied Computing (SAC)*, pages 1664–1667. ACM Press, 2015.

[4] M. García-Valdez, L. Trujillo, J. J. Merelo Guervós, F. Fernandez de Vega, and G. Olague. The EvoSpace Model for Pool-Based Evolutionary Algorithms. *Journal of Grid Computing*, 13(3):329–349, Sept. 2015.

[5] P. Salza, F. Ferrucci, and F. Sarro. elephant56: Design and Implementation of a Parallel Genetic Algorithms Framework on Hadoop MapReduce. In *Genetic and Evolutionary Computation Conference (GECCO Companion)*, 2016.