2017 25th Euromicro International Conference on Parallel, Distributed and Network-Based Processing

# Transparent execution of task-based parallel applications in Docker with COMP Superscalar

Victor Anton*, Cristián Ramón-Cortés*, Jorge Ejarque*, and Rosa M. Badia*†
*Barcelona Supercomputing Center (BSC), Barcelona, Spain
†Artificial Intelligence Research Institute - Spanish National Research Council (IIIA-CSIC), Barcelona, Spain
{victor.anton, cristian.ramonco, jorge.ejarque, rosa.m.badia}@bsc.es

***Abstract*—This paper presents a framework to easily build and execute parallel applications in container-based distributed computing platforms in a user transparent way. The proposed framework is a combination of the COMP Superscalar and Docker. We have built a prototype in order to evaluate how it performs by evaluating the overhead in the building, deployment and execution phases. We have observed an important gain compared with cloud environments during the building and deployment phases. In contrast, we have detected an extra overhead during the execution, which is mainly due to the multi-host Docker networking.**

***Keywords*-Cloud Computing, Linux Containers, Distributed Systems, Parallel Programming Models.**

## I. Introduction

Cloud Computing [1] has emerged as a computing paradigm where a large amount of computing capacity is offered on demand and only paying for what you use. This paradigm relies on virtualization technologies which offer isolated and portable computing environments called Virtual Machines (VMs) to perform the users' computations. Beyond the multiple advantages offered by these technologies and the overhead introduced during operation, the main drawback of these technologies in terms of usability is the management of VM images. The image creation process can take minutes for experienced developers, but it can be a complex and tedious work for scientist or developers without a strong background on these technologies.

To deal with these issues, a new trend in the Cloud Computing research has recently appeared. It proposes to substitute VMs managed by hypervisors with containers managed by engines such as Docker [2], which provide a more efficient layer-based mechanism that simplifies the image creation and provides a fast deployment mechanism. Benefiting from this new technology, the main contribution of this paper is to go a step further by integrating container engines with programming model runtimes, such as COMP Superscalar (COMPSs) [3] in order to provide an easy framework to create parallel applications and transparently deploy and execute applications in these container-based distributed platforms. From COMPSs, developers can benefit from straightforward programming model to parallelize applications from sequential codes and decoupling the application from the underlying computing infrastructure. Once the application is implemented, the COMPSs framework will automatically create the container images, deploy the required containers and execute the application distributing the computating tasks in the deployed containers.

The paper is distributed as follows: Section II describes related work, Section III presents the integration of COMPSs with Docker, Section IV presents the validation of the integration, and finally, Section V concludes the paper and gives some guidelines for future work.

## II. Related Work

Current Cloud providers and software stacks already provide basic services for image management and contextualization, which allow users to manually import, snapshot or create their VM images and configure the networking and security. However, as introduced in previous sections, this manual image modification can be a tedious work for complex applications. Therefore, recent research work has been focused on automating this process by adding new tools or services on top of the mentioned basic services. CloudInit [4] is one of the most used tools to automate the VM image creation. It consists of a package installed in the base image taht can be configured with a set of scripts which will be executed during the VM boot time. CloudInit is usually combined with DevOps tools like Puppet or Chef, where a manifest or a receipt is deployed instead of executing the configuration scripts (e.g. [5] or [6]). However, these solutions have a drawback, customizing the image at deployment time (installing a set of packages downloading files, etc.) can take some minutes. It can be assumable in the first deployment but not for adaptation where new VMs must be deployed in seconds. To solve this issue some services like [7] have been proposed to perform offline image modifications, in order to reduce the installation and configurations performed at deployment time.

In the case of containers, Docker already includes similar features to easily customize container images by defining a parent image and the required customization in a single file (Dockerfile). Due to the layered-based image system, parent and customized images can be reused and extended by applications achieving better deployment times. Cloud

2377-5750/17 $31.00 © 2017 IEEE
DOI 10.1109/PDP.2017.26

Providers, management stacks and workflow engines have stared to integrate container in their systems(e.g [8] or [9]). In our case, we propose integrating container engines with COMPSs to easily parallelize a sequential application and transparently deploy and execute applications in these container-based distributed platforms.

## III. Transparent Application Execution

### A. COMPSs and Docker Overview

COMPSs is a task-based programming model which aims to ease the development of parallel applications for distributed infrastructures. It is based on the idea that, in order to create parallel applications, the programmer does not need to be aware of all the underlying computer infrastructure details and does not need to deal with all the parallelism difficulties. In order to do this, the programmer has to specify which are the methods and functions that may be considered tasks and the direction of the task parameters. Given this sequential annotated implementation, the COMPSs runtime instruments the code to detect the defined tasks and build a task graph which includes the dependencies between tasks, constituting the workflow of the application. Then, it executes the application in parallel at task level managing the resources, scheduling tasks and handling data locality and tranfers. In addition, thanks to the abstracting layer that COMPSs provides, the same application and source code can be executed either in Clusters, Grids or Clouds.

Docker is an open platform for developing, shipping, and running applications. It provides a way to run applications securely isolated in a container. The difference between Docker and usual VMs is that Docker does not need the extra load of a hypervisor to run the containers and it uses an efficient read-only layered image system achieving lighter deployments. To improve Docker experience, several services and tools have been created. The relevant ones for this paper are: Docker-Swarm, Docker-Compose and Dockerhub. The first one, Docker-Swarm, is a cluster management tool for Docker. It merges a pool of Docker hosts enabling the deployment of containers in the different hosts with the same Docker API giving the impression to the user that it has a single, virtual Docker host. Docker-swarm is in charge of transparently scheduling containers on resources and managing the inter-host networking and storage. Docker-Compose is a tool to easily define complex applications which require to deploy multiple Docker containers.Finally, Dockerhub is a cloud-based image registry service that enables users to store and share their applications as docker images.

### B. COMPSs integration with Docker

Figure 1 provides an overview of the integration of COMPSs with Docker. From the user point of view, the only difference between running the application in the local machine or in the Docker infrastructure is the submission command. In a normal COMPSs application execution, users

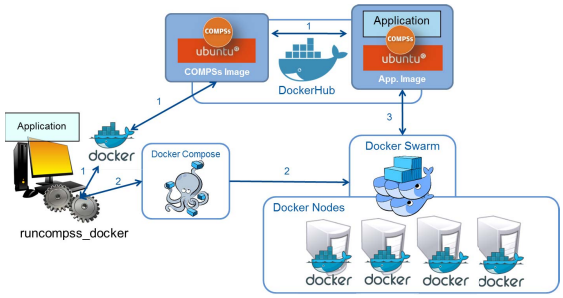



Figure 1. Integration of COMPSs with Docker

have to invoke the *runcompss* command followed by the application main class and arguments. This command loads the COMPSs runtime and starts the application execution. In the case that users want to run the application with Docker, they have to invoke the *runcompss_docker* command, similarly to the *runcompss* case, but adding some extra arguments to specify the Docker-Swarm manager IP address and port and how many containers must be deployed as computing resources. This command creates a docker application with the original COMPSs application, deploys it in the Docker infrastructure and executes the application. The *runcompss_docker* execution process is composed of two steps: the creation of Docker image for the application and the application execution in the container engines. The first step is done only once for each application, and the second step runs every time an application is executed.

In the first step, the script installs the application on top of the COMPSs base image and uploads it to Dockerhub so it can be easily shared between the cluster nodes. This COMPSs base image is a public Docker image located at Dockerhub which already contains a ready to the COMPSs runtime installed with all the required dependencies. Once the application image is created, the new layer created on top of the COMPSs base image is uploaded to Dockerhub. In this way, the different COMPSs applications deployed in docker will share the same COMPSs base layer. Therefore, deploying a new COMPSs application will only require to download the new application layer, which reduces the data to download from Dockerhub.

The second step is the deployment and execution of the COMPSs application in Docker containers. In this step, the script defines a Docker-Compose application by defining a master container, which will execute the main application and a set of worker containers which will start the COMPSs worker runtime to execute the parallel tasks. Despite the containers execute different parts of the application, both type of containers deploy the created application image. Once the application is defined, it uses Docker-Compose to deploy the containers and an overlay network in the docker cluster managed by Docker-Swarm. The application

execution starts once all the containers are deployed. In the case of the worker containers, they just wait for the master messages, and in the case of the master container, it executes the application as a normal COMPSs application configured to spawn the computing tasks in the deployed worker containers. When the application is finished, results are copied back to the user's machine and all containers are shut down and removed.

## IV. Experimentation

We have run a set of experiments to evaluate the integration of COMPSs with Docker with respect to the operation in the bare metal or a traditional Cloud. All the experiments have been done using the Chameleon Cloud infrastructure [10] which provide a configurable experimental environment for large-scale cloud research. On top of this infrastructure, we have set up three different environments: Bare-metal, Docker-Cluster and KVM-Openstack. The first scenario consists of a set of Bare-metal flavored nodes where we directly run COMPSs applications. The second scenario consists of a Docker Swarm cluster built on top of the same Bare-metal nodes, where each bare-metal node hosts a Docker Engine that will deploy the COMPSs applications' containers. Finally, the third scenario consists on an OpenStack managed Cloud with the same nodes virtualized with KVM. Each scenario uses up to 9 bare-metal nodes with 2 x Intel Xeon CPU E5-2670 v3 with 12 cores each and 128GB of RAM. When running a COMPSs application, one node, container or VM will run as a master, which manages the application execution, and the rest will run as workers. In all the cases, nodes, containers and VMs are defined to use the whole compute node (24 VCPUs and 128 GB of RAM) and deploy the same image (Ubuntu-14.04 with COMPSs 1.4).

The experimentation performed consists on the deployment and execution of two benchmark applications in the different environments. The first application consists of a blocked multiplication of two big matrices. They have been divided in 256 square blocks (16 x 16), each of them containing $2^{20}$ elements. It is quite I/O intensive because data dependencies between tasks require to transfer the different matrix blocks through the network as well as some disk utilization. In contrast, the second experiment is an embarrassingly parallel application without data dependencies which simply performs in parallel a series of trigonometric computations without exchanging any data. In this case the I/O utilization is mainly used by the messages exchanged by COMPSs to run the parallel computations.

In the case of the deployment evaluation, we have measured the time to perform the application deployments at the differnt environments and situations (when images are in the infrastructure, or not, etc.). For this experiment, we have not considered the bare metal since the computing nodes are already set-up and the installation of COMPSs and the applications must be done manually in all the nodes. Measurements for the rest of scenarios are summarized in Table I. In the case of Cloud, the process of customizing the VMs (deploy a with the VM base image, install the application and obtain a VM snapshot) takes around 100 seconds. Then, the VM deployment phase takes between 30 seconds, when the node has cached the image, or 98 seconds if image must be copied to the node. So, the total creation and deployment time in the case of a Cloud can take from 33 seconds up to 208 seconds. A similar issue happens with Docker, due to the layered image management, the creation and deployment time can take between 5 seconds, when all the layers are in the cluster, to 99.67 seconds, when no layers are available in the system. So, the Docker version significantly improves the deployment.

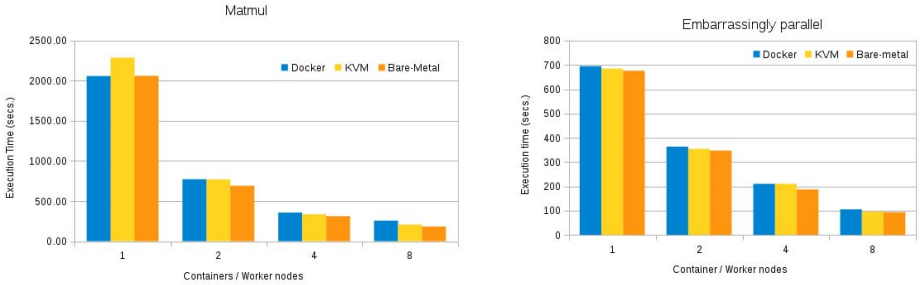
For both applications, we have measured the execution time using different number of nodes at the different environments. Measurements for the Embarrassingly Parallel and Matrix Multiplication benchmarks are depicted in Figure 2. In the case of the Embarrassingly Parallel, all platforms almost achieve the same performance (overheads are around 5%). This is because the overhead introduced by Docker and KVM in terms of CPU and memory management is relatively small. The difference is basically due to the multi-host networking used by the runtime to send the messages to remotely execute tasks in the workers. The default *overlay* network of Docker is performing worse than the *bridge* network of KVM. In the Matrix Multiplication case, we can see that Docker and bare-metal are performing similarly in this case of a single node, and KVM is performing a bit slower than Bare-metal (around 10%). This is because KVM has more overhead when managing disk I/O than Docker. However, when we increase the distribution of the computation (2,4,8 nodes), the multi-host networking overhead increases and becomes the most important source of overhead.

## V. Conclusion

In this paper, we have presented a method to integrate Container engines with parallel programming models, such as COMP superscalar (COMPSs), in order to create a framework to easily develop parallel applications and transparently deploy and execute applications in container-based distributed computing platforms. This integration consist on transparently converting COMPSs applications into Docker applications and execute them in Docker engines. It is is mainly focused on creating a Docker image with the COMPSs application and defining a Docker-Compose application which automatically deploys and executes the required containers to execute the application. This integration brings several benefits for developers. COMPSs provides a straightforward methodology to parallelize applications from sequential codes, while Docker provides an efficient packaging an deployment tools to distribute applications.

| Phase | KVM-OpenStack | | | Docker | | | | |
|---|---|---|---|---|---|---|---|---|
| | Action | Time | | Action | Time | | | |
| | | w/o Image | Image Cached | | w/o Images | w Ubuntu | w COMPSs | w App. |
| Build | Base VM deployment | 33.58 s. | N/A | Image Creation | 73.87 s. | 68.88 s. | 15.48 s. | N/A |
| | App. Installation | 15.45 s. | N/A | Image upload | 8.66 s. | | | N/A |
| | Image Snapshot | 60.36 s. | N/A | | | | | |
| Deployment | VM deployment | 83.68 s. | 18.18 s. | Image Download | 12.39 s. | | | N/A |
| | VM boot | 15.09 s. | | Container deployment | 4.75 s. | | | |
| Total | | 208.16 s. | 33.27 s. | | 99.67 s. | 94.68 s. | 41.28 s. | 4.75 s. |

Table I
APPLICATION DEPLOYMENT




(a) Matrix Multiplication (With Data dependencies) (b) Embarrassingly Parallel (No data dependencies)

Figure 2. Execution Overhead and Scalability

Besides, we have run a set of experiments to investigate how this integration performs in terms of deployment and execution time. The results have been compared with other alternatives (bare-metal and KVM/OpenStack cloud). In terms of deployment, we have seen a significant gain compared with VM deployment. In terms of execution, we have seen that the system performs similarly than bare-metal or KVM for applications without or small data dependencies. However, the drawback of Docker containers appears with the usage of multi-host networking is important enough. In this situation, Docker has a larger overhead than KVM.

As future work, we are going to evaluate experimental alternatives for Docker multi-host networking in order to test if COMPSs with Docker can perform better than KVM in all situations. Moreover, we are going to extend our work to support other container engines such as Kubernetes, as well as container services provided by major vendors.

## ACKNOWLEDGMENT

This work is partly supported by the Spanish Government through contracts SEV-2015-0493, TIN2015-65316-P, by the Generalitat de Catalunya under contracts 2014-SGR-1051 and 2014-SGR-1272, and by the European Union under grants 676556 (MuG Project) and 690116 (EUBra-BIGSEA Project). Results presented in this paper were obtained using the Chameleon testbed supported by the NSF.

## REFERENCES

[1] M. Armbrust *et al.*, "Above the clouds: A berkeley view of cloud computing," *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2009-28*, 2009.

[2] D. Merkel, "Docker: lightweight linux containers for consistent development and deployment," *Linux Journal*, vol. 2014, no. 239, p. 2, 2014.

[3] R. M. Badia *et al.*, "Comp superscalar, an interoperable programming framework," *SoftwareX*, vol. 3, pp. 32–36, 2015.

[4] "Cloud-init ," Web page at https://launchpad.net/cloud-init, (Date of last access: 15th September, 2016).

[5] D. Armstrong *et al.*, "Contextualization: dynamic configuration of virtual machines," *Journal of Cloud Computing*, vol. 4, no. 1, p. 1, 2015.

[6] D. Bruneo *et al.*, "Cloudwave: Where adaptive cloud management meets devops," in *2014 IEEE Symposium on Computers and Communications (ISCC)*. IEEE, 2014, pp. 1–6.

[7] J. Ejarque *et al.*, "Service construction tools for easy cloud deployment," in *7th IBERIAN Grid Infrastructure Conference Proceedings*, p. 119.

[8] "Nova-Docker driver for OpenStack ," Web page at https://github.com/openstack/nova-docker, (Date of last access: 15th September, 2016).

[9] C. Zheng and D. Thain, "Integrating containers into workflows: A case study using makeflow, work queue, and docker," in *Proceedings of the 8th International Workshop on Virtualization Technologies in Distributed Computing*. ACM, 2015, pp. 31–38.

[10] "Chameleon Cloud Project," Web page at https://www.chameleoncloud.org, (Date of last access: 15th September, 2016).