Container-based Virtual Elastic Clusters

Carlos de Alfonso[a,*], Amanda Calatrava[a], Germán Moltó[a]

[a] *Instituto de Instrumentación para Imagen Molecular (I3M)*
Centro mixto CSIC - Universidad Politécnica de Valencia - CIEMAT
Camino de Vera s/n, 46022, Valencia

[*] Corresponding author: Tel. +34963877356
Email address: caralla@upv.es (Carlos de Alfonso)

Abstract

eScience demands large-scale computing clusters to support the efficient execution of resource-intensive scientific applications. Virtual Machines (VMs) have introduced the ability to provide customizable execution environments, at the expense of performance loss for applications. However, in recent years, containers have emerged as a light-weight virtualization technology compared to VMs. Indeed, the usage of containers for virtual clusters allows better performance for the applications and fast deployment of additional working nodes, for enhanced elasticity. This paper focuses on the deployment, configuration and management of Virtual Elastic computer Clusters (VEC) dedicated to processs scientific workloads. The nodes of the scientific cluster are hosted in containers running on bare-metal machines. The open-source tool Elastic Cluster for Docker (EC4Docker) is introduced, integrated with Docker Swarm to create auto-scaled virtual computer clusters of containers across distributed deployments. We also discuss the benefits and limitations of this solution and analyse the performance of the developed tools under a real scenario by means of a scientific use case that demonstrates the feasibility of the proposed approach.

Keywords: Computing, Containers, Cluster Computing, Elasticity

1. Introduction

eScience involves the execution of complex HTC (High Throughput Computing), HPC (High Performance Computing) applications and long-running workflows. This requires a significant amount of computing power and memory capacity that can be only obtained via distributed computing. Indeed, large-scale Distributed Computing Infrastructures (DCIs), such as the European Grid Infrastructure (EGI)[1] have been tremendously successful in supporting the computational requirements of many scientific communities across Europe [1, 2]. However, one of the main limitations of Grid infrastructures is that applications have to be ported to the execution environments provided by the machines involved, what results in a rigid structure composed by several Virtual Organizations (VOs) that support a set of applications. This inability to provide customized execution environments for applications is addressed by Cloud Computing by means of Virtual Machines (VMs) that encapsulate the Operating System (OS) together with the user application and its dependences in a Virtual Machine Image (VMI) that can be run on a physical machine by means of a hypervisor.

Indeed, the ability to provide ubiquitous, on-demand network access to a set of configurable computing resources, according to the NIST definition [3] of Cloud Computing, has paved the way for the rise of many public Cloud providers (such as Amazon Web Services (AWS)[2], Microsoft Azure[3] or Google Cloud Platform[4]), different Cloud Management Frameworks (such as OpenNebula or OpenStack) and even initiatives to create large-scale community Clouds (e.g. EGI Federated Cloud[5]). Cloud computing has provided researchers with access to unprecedented customizable computing resources, either on-premises or on public Clouds. However, these computing resources still require a coor-

[1] European Grid Infrastructure: `http://www.egi.eu`

[2] Amazon Web Services: `https://aws.amazon.com`

[3] Microsoft Azure: `https://azure.microsoft.com`

[4] Google Cloud Platform: `https://cloud.google.com`

[5] EGI Federated Cloud: `https://www.egi.eu/federation/egi-federated-cloud/`

dinated use for applications to efficiently use them. For that, Local Resource Management Systems (LRMS) such as Torque [4], SLURM [5] or HTCondor [6] are job schedulers that are commonly used to dispatch jobs across nodes [7]. Indeed, computing clusters are still widely-used computing facilities to support the execution of many types of applications.

A scientific computing cluster is a type of parallel or distributed processing system, which consists of a collection of interconnected stand-alone computers working together as a single integrated computing resource [13]. The access to a scientific cluster is usually made by means of a SSH connection to a "front-end" computer, and the users submit tasks to a middleware that will coordinate the working nodes to run these tasks. All the computers usually share filesystem to ease the distribution of the applications and the data that they need.

Virtual Elastic scientific computer Clusters (VEC) deployed on Cloud infrastructures have introduced many benefits when compared to physical clusters, as we addressed in our previous work [8], avoiding upfront investments and the ability to adapt the execution environment to the applications (and not viceversa). This work was later extended to create EC3[6] [9] an open-source tool to create self-managed cost-efficient virtual hybrid elastic clusters across Clouds that is currently offered as a free online service, being used for scientists to provision their own clusters on public, on-premises and federated Clouds.

In the quest for increased performance with respect to virtualisation techniques, Linux containers appeared as a lightweight alternative to VMs. Linux containers enable to run multiple isolated processes in a host without the overhead caused by the hypervisor layer introduced by VMs. While hypervisors provide hardware abstraction, container-based virtualization is characterised by multiple isolated user spaces running at the operating system level (see Figure 1). This provides process isolation at a fraction of the overhead introduced by the hypervisor. Container-based virtualization proved to be an alternative to traditional hypervisor-based systems, as it reduces the overhead caused by VMs

[6]EC3 (Elastic Cloud Computing Cluster): `http://www.grycap.upv.es/ec3`

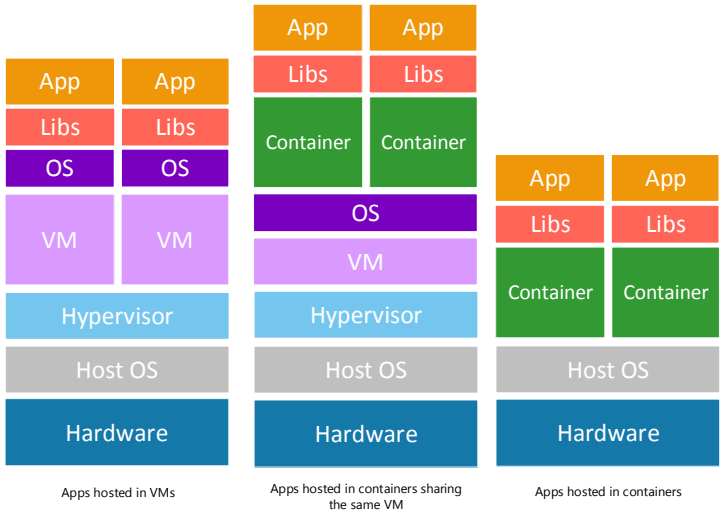


Figure 1: Virtual Machines and Containers possible architectural configurations.

in CPU, memory and storage, as described in [10] and [11]. Linux containers can be run on top of VMs to achieve multi-tenant isolation using the VM as the boundary of security and containers as the boundary of resource allocation to applications. However, the main benefits of containers arise when used on bare metal, in order to obtain increased performance compared to VMs. Among the different existing container platforms, Docker[7] stands out as a software containerization platform that can encapsulate an application in a complete filesystem that contains all the dependences required to be executed (code, runtime, system tools and libraries, etc.). This guarantees portability across multiple platforms, regardless of the execution environment.

Our hypothesis is that container-based technology can be effectively integrated with cluster-based computing to create virtual computer clusters of Docker containers with the very same functionality as virtual clusters of VMs, and physical clusters of PCs, but with enhanced capabilities that include: i)

[7]Docker: `https://www.docker.com`

improving the performance of resource-intensive applications that will run isolated on bare metal; ii) improving the elasticity of the cluster, by reducing the time required to spawn and terminate additional containers and iii) supporting customised execution environments via low-footprint images.

Therefore, this paper introduces an architecture to deploy container-based virtual scientific computer clusters that feature automated elasticity and the ability to provide customised virtual execution environments across a bare-metal backend on which containers managed by a Container Orchestration Platform (COP) are executed. Several computer clusters customised for the execution of different scientific applications can be provisioned to share the same physical computing backend. This provides increased resource utilisation and performance while maintaining isolation across workloads coming from different clusters.

To this aim, this paper describes EC4Docker[8], an open-source tool to deploy, configure and manage container-based virtual computer clusters that can be run on bare-metal nodes (as well as on VMs). These virtual computer clusters expose the very same user interfaces expected by users (accessed via SSH, supporting a LRMS, etc.) but they are completely backed by Docker containers that are dynamically deployed, depending on the workload, across a distributed Docker Swarm [12] backend that can be deployed either on bare metal or on public and on-premises Clouds.

After the introduction, the remainder of the paper is structured as follows. First, section 2 introduces background information and covers the state of the art related to containers, revising existing tools, performance studies and clustering solutions of containers. Next, section 3 exposes and analyses the proposed architecture to deploy these container-based virtual computer clusters. Then, section 4 addresses different scenarios in which the proposed solution is evaluated and analyses the significant benefits of these approach. Finally, section 5 summarises the paper and points to future work.

[8]EC4Docker is available in `https://github.com/grycap/ec4docker`

2. Background and Related Work

According to Buyya [13], a computer cluster is a type of parallel or distributed processing system, which consists of a collection of interconnected stand-alone computers working together as a single integrated computing resource. The key components of a cluster include:

(i) Multiple Computers. Typically one of them (named the "front-end node") acts as an entry point to the computer cluster and the others execute the jobs (named the "working nodes").

(ii) Operating Systems (OS). In scientific computing, the most common operating systems are Linux or Unix-based.

(iii) Interconnection network. The computers interact among them through a local network. There may exist different networks specialised for different tasks (e.g. data, parallel processing, etc.) based on different technologies for computers to communicate (e.g. Myrinet, GbE, 10GbE, etc.).

(iv) Cluster middleware. The cluster middleware, also known as LRMS, is a set of tools to use the cluster as a single computing entity. These tools carry out the whole lifecycle of executing a job in the cluster (e.g. staging the files in the working nodes, starting the applications, retrieving the resulting files, etc.). Some examples of well known LRMS are Torque, SLURM, or LSF.

(v) Parallel programming environments. Applications typically use well-known libraries to communicate between processes. Some examples are Open-MPI, LAM/MPI or MPICH, which support the Message Passing Interface (MPI) standard. These libraries are usually optimised for the specific network interfaces (e.g. SCI, Myrinet, etc.).

(vi) Applications. These are the user applications executed in the computer cluster.

The main interface employed by the users of the cluster is an interactive session to the front-end node in order to submit jobs to be executed on the working

nodes [14]. Indeed, computer clusters used to be huge physical infrastructures, but advances in virtualization technologies and Cloud computing paved the way for Virtual Clusters (VCs) to appear. A VC is comprised of VMs and a virtual networking environment. The other components in a VC are the very same that those used in the physical cluster. These VCs can be deployed in on-premises infrastructures or in commercial public Clouds.

A VC relies on VMs even if they are not used (i.e. they are idle). These idle virtual working nodes are a problem in a Cloud environment because (a) in case the cluster is deployed in an on-premises infrastructure, other users cannot take advantage from the unused resources allocated to the VC, or (b) in case the cluster is deployed in a public Cloud, the unused resources result in an economic cost for the user. An Elastic Virtual Cluster (EVC) avoids wasting either resources or money, by destroying the idle working nodes and deploying them again when they are needed. In order to implement an EVC, an elasticity manager is required to take care of creating or destroying the working nodes, depending on the workload.

The work described in this paper is a step forward on computer cluster virtualization, that builds on container-based virtualization to reduce the performance penalty introduced by VMs. The goal for a container-based EVC is to provide the users with computer clusters to be used as if they were physical computing clusters, with the added value of using containers instead of VMs. Therefore, the requirements for the container-based EVC is to preserve the very same environment and usage patters that are commonly used in this computing platforms, i.e. the software stack: the OS, the cluster middleware, the parallel environments and the applications, as shown in Figure 2.

The next section includes a review of related works about the different technologies that lie within the scope of this work.

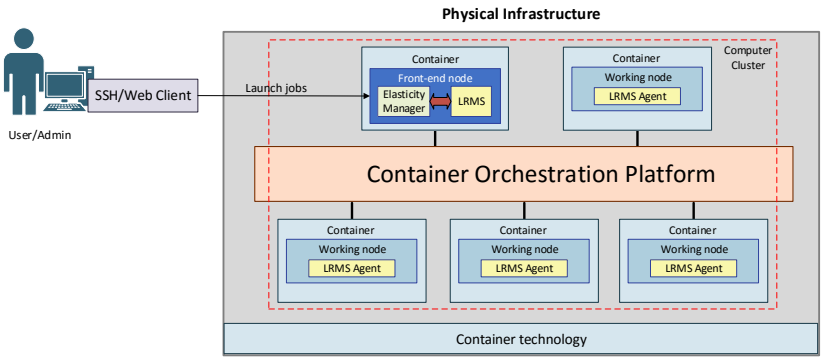


Figure 2: Generic architecture to deliver container-based virtual elastic computer clusters deployed on a computing infrastructure managed by a Container Orchestration Platform.

2.1. Related work

2.1.1. Containers

Container technologies have gained significant momentum in the last years, introducing changes in the way applications are built, shipped, deployed, and instantiated [15], [16]. There exists different software available to create Linux containers, as is the case of Linux Containers (LXC) [17] and LXD [18], rkt [19], OpenVZ [20], Linux-VServer [21] and Docker [22]. In particular, Docker turned containers into a mainstream technology, contributing: i) Docker Hub, a global shared repository of Docker containers; ii) a procedure to create Docker images out of Dockerfiles and iii) the usage of a layered file system that reduces the footprint of Docker images. Docker containers use *cgroups*, a feature in the Linux kernel that allows to constrain the resources (e.g. CPU, memory and network) consumed by a process together with *namespaces* to provide processes with their own view of the system. In our case, the containers will correspond to the working nodes that compose the VC.

2.1.2. Container Orchestration Platforms

The ecosystem of applications around Docker has exploded in the last years [23], with contributions in many areas such as Continuous Integration/Continuous

Delivery (CI/CD), application packaging and Container Orchestration Platforms (COPs). Indeed, there are many applications to manage the execution of containers across multiple hosts. For example, Kubernetes [24] is an open source orchestration system for Docker containers. It handles scheduling onto nodes in a compute cluster and actively manages workloads to ensure that their state matches the user's declared intentions. The scheduling in Kubernetes is based in *Pods*. These are groups of containers that are deployed and scheduled together. Pods form the atomic unit of scheduling in Kubernetes, as opposed to single containers in other systems. Containers within a pod share an IP address, and labels can be used to identify each group of containers. Apache Mesos [25] can be used to deploy and manage applications inside containers in large-scale clustered environments. The architecture of Mesos is designed to be high-available and for that uses ZooKeeper. Mesos, in combination with a job system like Marathon [26] or Chronos [27], takes care of scheduling and running jobs and tasks, that can be run in containers or directly in the nodes of the cluster. Finally, Docker Swarm [12] represents the native clustering approach proposed by Docker, which "provides native clustering capabilities to turn a group of Docker engines into a single, virtual Docker Engine". This way, a container-based VC can be easily created on top of virtual or physical resources. The architecture of Docker Swarm consists of hosts running a Swarm agent (working nodes) and one host running a Swarm manager. The manager is responsible for the orchestration and scheduling of containers on the hosts. Moreover, Docker Swarm can be run in a high-availability mode where either etcd, Consul or ZooKeeper is used to handle fail-over to a back-up manager. We opted for Docker Swarm due to its easy integration with the Docker CLI (Command-Line Interface).

Notice that COPs are used to manage the execution of containers in a cluster. The user describes the container, and the COP selects which of the physical host is going to perform the execution of the container. Therefore, these tools represent for containers a similar concept than a LRMS (e.g. Torque, SLURM, etc.) is for jobs in a computer cluster.

Notice that one could use the interfaces provided by a COP to directly deploy

containers that run their jobs on a set of computing resources. However, this approach would be disruptive for traditional users of computer clusters since their usage patterns would significantly change. They would have to use client-side tools to interact with such COPs and deal with data staging in the COP, instead of performing an interactive session via SSH to the cluster. Instead, the clusters deployed via EC4Docker maintain the very same user experience and interfaces exposed by traditional computer clusters (e.g. SSH-based access to the front-end node).

2.1.3. Reducing overhead of VMs using Containers

There are studies in the literature that analyse the overhead of containers for the execution of applications. In [10], the authors explore the performance of traditional VM deployments and contrast them with the use of Linux containers (using Docker). Several benchmarks are used to demonstrate that containers result in equal or better performance than VMs in terms of CPU, memory and storage. The study covered in [28], analyzed the performance of three well known open-source tools (KVM, OpenVZ, and Xen) in the context of HPC. The results showed that the solution that offers near native CPU and I/O performance was OpenVZ. Other works in the literature have also analyzed the performance of containers to execute scientific applications and workflows, such as [29] and [30]. Skyport [31] utilizes Docker containers to execute scientific workflows instead of VMs, reducing the overhead caused by VM virtualization. Also, analysis of the requirements of the applications to be executed in containers have been performed [32]. Because container-based virtualization works at the operating system level, all instances (containers) share the same operational system kernel. That is why container-based virtualization has a weaker isolation when compared to hypervisor-based virtualization [33]. In order to guarantee the resource isolation between the host system and the containers running on, such a system implements kernel namespaces. However, using containers for security isolation might not be a good idea [34]. The only way to have real isolation with Docker is to either run one container per host, or one container per VM,

at the expense of a performance overhead. Nevertheless, for security reasons, it might be worth sacrificing the performance of a pure-container deployment by introducing a VM to obtain true isolation.

Containers can run on VMs too, although such double virtualization imposes performance overheads. In [35] authors investigate container-based technology as an efficient virtualization technology for running high performance scientific applications. They used Docker containers and VMs created using OpenStack to execute a molecular modeling simulation software. Results show that container-based systems are more efficient in reducing the overall execution times for HPC applications, because they can be deployed in a remarkable minor time and have better memory management for multiple containers running in parallel.

2.1.4. Virtual computer Clusters

Concerning the use of VC, several well-known tools already exist in the literature to deploy them, such as StarCluster [36], Elasticluster [37] and EC3 [9], but all of them are based on the deployment of VMs. Concerning the creation of VCs based on containers, studies like [38] analyzed and compared some of the container technologies available to the community (Linux-VServer, OpenVZ and LXC) from the point of view of MapReduce workloads, executing several benchmarks to test their performance and manageability. The results show that container-based systems reached near-native performance though LXC offers the best relationship of performance and isolation. The study covered in [39] present the results of deploying Docker containers in a cluster environment when compared to the KVM hypervisor and an evaluation of its suitability as a runtime for high performance parallel execution. The results showed that containers can be used to tailor the runtime environment for an MPI application without compromising performance, and provide better Quality of Service for users of scientific computing. The developers of a Linux-VServer address in [40] a container-based cluster management platform in which Docker and HTCondor work together to execute scientific workflows. The results obtained from executions of a Monte-Carlo simulation showed that Docker had a near native

performance comparing with a hypervisor-based virtualization solution.

To our knowledge, there are no works in the literature that feature the adoption of Docker containers to create VCs that provide users with the very same execution environment (e.g. LRMS, client tools) typically available in both physical clusters and virtual clusters of VMs. This pioneer approach allows users to access well-known computing facilities, i.e. clusters of PCs, on top of the lightweight virtualisation provided by containers in order to take profit from enhanced performance and fast elasticity.

3. Elastic Cluster for Docker (EC4Docker)

EC4Docker is an open-source tool that deploys Docker container-based Virtual Elastic computer Clusters (CVEC). The cluster delivered by EC4Docker consists of a Docker container that acts as the front-end node of the cluster, and a set of containers that act as the working nodes. The front-end container behaves as a regular front-end in a cluster: it is accesible by SSH, has installed a LRMS such as Torque or SLURM, and it shares its file system to the working nodes using NFS (Network File System). The working nodes of the EC4Docker cluster are also containers that behave like regular working nodes in a cluster: they are accesible from the front-end using password-less SSH, they are integrated in the LRMS, and they mount the shared file system.

The novelty of EC4Docker is that the front-end of the cluster is able to create and to destroy the internal nodes depending on the workload. This ability is possible due to: i) the integration of the CLUES[9] [41] elasticity manager that decides when to power on or off the internal nodes and ii) a plugin for CLUES that has been developed for EC4Docker, that makes it possible to translate the commands to power on and off of the internal nodes into the proper Docker instructions that create and destroy Docker containers. This plugin takes advantage from the ability of Docker to be used remotely by exposing its API

[9]CLUES: `https://github.com/grycap/clues`

through a standard TCP/IP socket.

The concept of a container-based cluster as-is may be useful for prototyping, as the containers are conceived to be ran in a specific host. However, in the case of EC4Docker, it is integrated with Docker Swarm, which behaves as a scheduler that manages a set of Docker hosts as a single entity. It works together with a discovery service, such as Consul[10], that provides high availability to the underlying Docker Swarm cluster. Using Docker Swarm, when a Docker container is created, it is deployed in any of the Docker hosts managed by the Swarm. Using this combination of EC4Docker and Docker Swarm, it is possible to deploy the containers that build the CVEC across multiple hosts which, by the way, can be either physical or VMs. It is important to point that other COPs could be employed instead of Docker Swarm, such as Kubernetes or Apache Mesos as well as managed services for the deployment of containers such as Amazon EC2 Container Service[11].

3.1. Features of the Container-based Virtual Elastic Computer Cluster

As stated earlier, using EC4Docker, the users are delivered a computer cluster with the tools that they typically use, and they do not need to change the way of interacting with the cluster. They access the cluster using SSH, where they find the LRMS to which jobs can be submitted as usual. The LRMS is not aware of any container and the applications require no modifications.

However, even experienced users in traditional computing clusters can benefit from the CVEC, because these are useful to create the specific execution environment for their applications. Docker containers are commonly employed to ease the distribution of applications: using *Dockerfiles* in Docker, users can create the container images that include their application along with the required libraries, the most appropriate OS distribution, etc. Starting from that Docker image, the administrator will include the EC4Docker *Dockerfiles* that

[10] Consul: `https://www.consul.io`

[11] Amazon EC2 Container Service: `https://aws.amazon.com/es/ecs/`

will create the EC4Docker CVEC that will be delivered to the user.

Using this approach, although the underlying infrastructure is shared by all the CVEC, different configurations can be employed. For example, a cluster based on Ubuntu 16.04 and the Torque LRMS can coexist and share the same underlying computational resources with a Scientific Linux cluster whose jobs are scheduled by SLURM. It is important to point out that this feature can be very beneficial for the execution of software applications that are incompatible with each other, without needing to physically isolate the resources. Therefore, the bare-metal physical nodes are shared by all the clusters deployed in the infrastructure, where the container-based working nodes will be deployed to execute the jobs of each cluster.

EC4Docker is not only useful for CPU-oriented applications. In case the applications require access to specific devices, such as GPGPUs, it is possible to instruct EC4Docker to allow the Docker containers to access these devices. On the one hand, in the case of homogeneous configurations where all the physical nodes have a GPGPU, EC4Docker can be instructed to automatically mount that device inside the container to expose it to applications. In this case, EC4Docker will use the Docker mechanisms to enable the applications to use the GPGPUs available in the physical hosts. For this, the container has to support the specific libraries and drivers required to use the GPGPU. On the other hand, in the case where only a subset of the physical nodes have a GPGPU, rCUDA [42] can be used in order to turn those nodes into servers that provide GPU services to the container nodes that actually execute the applications. The applications do not require source code modification since the rCUDA runtime takes care of the details of routing requests to the specific hardware device.

3.2. Behaviour of a container-based virtual elastic cluster

Figure 3 describes the designed architecture employed to deploy CVECs on top of a physical infrastructure, as an instantiation of the general architecture shown in 2. Therefore, the workflow to create the CVEC follows the next steps:

1. *Preparation of the Docker images.* The preparation of a CVEC starts with

the creation of the Docker images that will be used to create the front-end and the working nodes, and its instrumentation using the EC4Docker Dockerfile fragments.

2. *Creation of a network for the container.* The CVEC needs a network for containers to communicate. In case of using Docker Swarm, an *overlay* network that spans across the different sites is required. This overlay network enables different hosts to become part of a swarm and assign non-overlapping IP addresses to their containers to enable communication among them. There is the option of using a single overlay network shared among all the containers from all the CVECs, or to create per-cluster networks in order to isolate the different CVEC. The overlay network is used to virtualize the interconnection network for the creation of the CVEC.

3. *Creation of the CVEC.* The creation of the cluster consists of deploying the container that will act as the front-end of the CVEC. Since we want the computer clusters to span across multiple hosts, the request to create the container will be submitted to the Docker swarm front-end. A container will be instantiated out of a Docker image created from the EC4Docker Dockerfiles, which include an installation of CLUES and the LRMS chosen by the end-user (SLURM or Torque). The containers are used to virtualize the working nodes for the creation of the CVEC.

4. *Enable external access to the cluster.* In order to access the cluster using SSH, the IP address of the front-end node of the CVEC is required. However, the IP addresses in the Docker swarm cluster will be private to the overlay network for the cluster and, therefore, they are not accesible from outside networks. To solve this problem, we use IPFloater[12], a tool able to redirect the traffic from a public IP to a private IP inside a local area network (LAN), thus, simulating the floating IPs offered by OpenStack [43].

[12]IPFloater is available at `https://github.com/grycap/ipfloater`

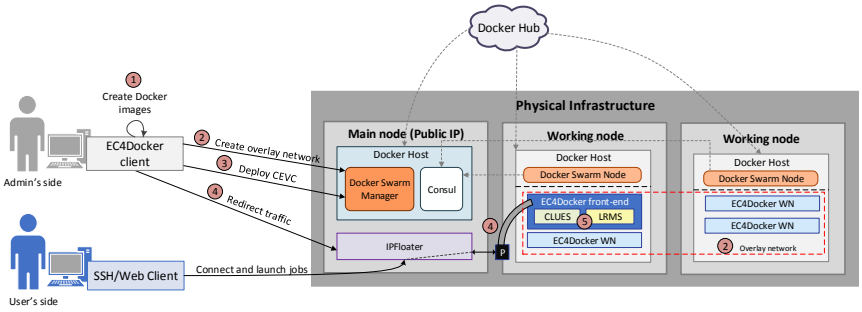


Figure 3: Architecture of a container-based virtual elastic cluster deployed on top of a physical infrastructure and managed with Docker Swarm and EC4Docker.

Once the workflow has finished, the user is provided with the IP address of the front-end node of the CVEC. Then, the end users can connect to the cluster via SSH or by means of a web browser (in case of accessing a web application like the Galaxy Portal [44]) and submit their jobs to the selected LRMS as they would do with a physical cluster.

The CVEC deployed using EC4Docker dynamically manages the size of the cluster, with the novelty of running the jobs that are going to be executed in the container-based working nodes, instead of using the traditional VM-based working nodes. This way, jobs will enjoy the advantages of light-weight virtualization with a reduced overhead in CPU and memory.

The self-managed elasticity is carried out by CLUES (step 5 in Figure 3), that forms part of the container image used by EC4Docker to deploy the front-end of the cluster. CLUES running inside the EC4Docker container detects job submissions to the LRMS in the container-based cluster. Then, if there are no available nodes to satisfy the requirements of the job, it requests an EC4Docker node container to the Docker Swarm Manager. This container will be deployed by Docker Swarm in one of the bare-metal nodes that compose the infrastructure, and will act as a container-based working node of the computer cluster, automatically integrated in the LRMS. The EC4Docker node container will be also connected to the overlay network specifically created for the CVEC,

interconnecting the new container with the rest of the CVEC.

As we have mentioned, the scheduling of the location of the containers that represent the cluster is carried out by Docker Swarm. Docker Swarm works with rankings to decide where to execute the container. The node with the highest ranking is the one that is chosen to run the new container. The policies offered by Docker Swarm are: *spread* (default), *binpack* and *random*. The first two policies care about the number of containers deployed in the node and the CPU and RAM free for each node, while the latter policy (*random*) simply returns a random value for each node. Through the *spread* policy, the node chosen to host the new container depends on the number of containers running on the node, regardless of their status. With the same resources (CPU and RAM), the node that has fewer containers will run the new container. The *binpack* policy, on the other hand, tries to pack the containers in a node, trying to leave free enough space in other nodes to hold containers with higher requirements. Thus, it avoids fragmentation. It is noteworthy that, for all the policies, if all nodes get the same ranking, the election is performed randomly.

3.3. Elasticity Rules

As stated earlier, elasticity in EC4Docker is managed by CLUES. This software implements different policies that aim at balancing the trade-off that arises when trying to minimize the waiting time for the jobs (which involves a larger number of available nodes) and the minimization of the infrastructure cost, which involves a reduced number of nodes, which generate a cost in electricity (for physical infrastructures) or in resources (for public cloud providers). In the context of containers, the creation of a container results in less available resources for the subsequent containers deployed on the same host. Therefore, it is important to submit the containers only when they are really necessary.

The policies implemented by CLUES can be divided in two groups: the policies used to decide when to increase the capacity of the cluster (scale-out) and those used to decide when to decrease the size of the cluster (scale-in). Regarding the scale-out policies, CLUES can interact with the LRMS at two

levels. On the one hand, it intercepts the submitted jobs before they reach the LRMS. On the other hand, CLUES also monitors the queued jobs at the LRMS to check if these jobs require additional nodes to be added to the cluster. The policies available are:

- **1:1 start**. For each job launched, if no working nodes are available for its execution, then a new node is deployed. Therefore, the jobs will wait for the deployment of the node before they start their execution.

- **Group-based start**. Every time a new node is required, a group of them are started. This policy assumes a workload model in which as soon as a job reaches the LRMS, there is a high probability that other subsequent jobs will be submitted in a short period of time. By over-provisioning a larger number of nodes, the waiting time of the subsequent jobs will be reduced.

In order to decide when to shutdown a node (scale-in policies), the strategy is to remove a node from the computer cluster when it has been idle for a specified amount of time. The selection of this time depends on the workload of the computer cluster and it is important to achieve a good trade-off between the used resources and the waiting time of the jobs. These are the available strategies:

- **Queued jobs**. Idle working nodes are terminated when there are no pending jobs in the LRMS.

- **Delayed shutdown**. Idle working nodes are terminated after a certain amount of configurable time. This is of interest when using public Clouds that bill by the hour, where idle nodes are kept available for job executions before the hour expires, even if no jobs are available to be executed at the moment.

- **Keeping some nodes always active**. The computer cluster will have a set of nodes deployed waiting for jobs. This way, the computer cluster tries to prevent incoming jobs from waiting while nodes are started.

4. Case study

In order to assess the effectiveness of the self-managed CVECs deployed with EC4Docker, we present a case study based on a bioinformatics community of users that need to execute several scientific tasks for their research. In particular, the application used is MrBayes [45] (Bayesian Inference of Phylogeny). MrBayes is a program for Bayesian inference and model choice across a wide range of phylogenetic and evolutionary models. MrBayes uses Markov Chain Monte Carlo (MCMC) methods to estimate the posterior distribution of model parameters. MrBayes has several dependencies in order to work properly, like an MPI implementation or the Beagle library. For this, we installed, among others, OpenMPI together with the *gcc* compiler. The case study also analyzes the performance of containers comparing to VMs, trying to prove the advantages of light-weight virtualization in contrast with traditional virtualization based on VMs.

The overall scenario consists of three different executions of the same job pattern submission (represented in Figure 4 (a)) on two different computing scenarios, but on top of the very same physical resources. On the one hand, scenario a) involves a container-based virtual computer cluster managed by EC4Docker. All the containers submitted during this execution were limited to 1 CPU and 1 GiB of RAM. On the other hand, scenarios b) and c) involve a VM-based virtual computer cluster deployed on an OpenNebula on-premises Cloud by means of EC3. Each one is configured with different idle times to trigger the scale-in policy: scenario b) is configured with a maximum value for idle nodes of 1800 seconds (30 min.), and scenario c) will power off nodes that were idle for more than 600 seconds (10 min.). Each VM deployed has 1 CPU and 1 GiB of RAM. The VMI employed is based on Ubuntu 14.04 LTS. In this case, two different executions for each configuration were carried out, one in which the software is dynamically deployed on vanilla VMs and the other in which the software (SLURM, OpenMPI, NFS, MrBayes and its dependencies) is pre-installed in the VMI, thus reducing the time for contextualization, i.e.,

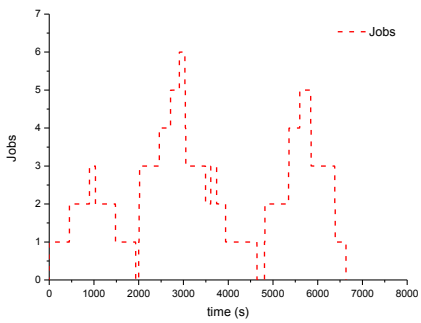
installation and configuration of the software applications. This last option was the one chosen to compare with the execution of the container-based cluster, since Docker containers are created out of pre-configured Docker images.

The physical infrastructure used to deploy the case study is the same for both scenarios for the sake of a fair comparison. It comprised eight physical nodes with a total of 224 cores (28 cores per node), 512 GB of RAM (64 GB of RAM per node) and a shared storage system of 16 TB. For the scenario a) we deployed Docker Swarm and the main node includes the IPFloater tool in order to associate a public IP to each container-based front-end. In scenarios b) and c), an OpenNebula 4.8.0 on-premises Cloud deployment is used.

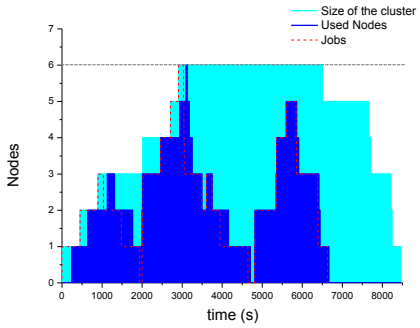
The limit size of the cluster was fixed to 6 nodes in all scenarios. A total of 15 Bayesian tasks with an average duration of 17.5 minutes is executed for each test. The dataset employed is *cynmix.nex* [46], a partitioned dataset consisting of data from four genes and morphology for 30 taxa of gall wasps and outgroups. The number of generations has been fixed to 170.000. The following subsections describe and analyze the obtained results for this case study.

4.1. Results

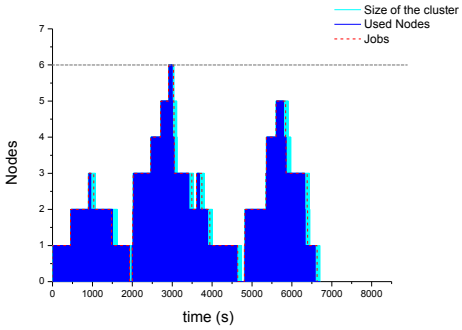
First, we analyzed the time differences in the deployment and contextualization processes for both containers and VMs used in our case study. Table 1 shows the average times for the deployment, configuration and execution times for the three scenarios. As we expected, the total average times for both the front-end (FE) and working nodes (WN) were considerably higher with VMs, even if we use a preconfigured VMI with SLURM, NFS, OpenMPI and MrBayes dependencies previously installed. In the last case, it was still necessary to configure the SLURM configuration files, NFS system, and the application MrBayes, that takes an average time of [335-340] seconds in the case of the front-end and [284-285] seconds for the working nodes. Even so, the time consumption during the contextualization process was reduced significantly by starting from a preconfigured VMI (about 65%). However, preparing a customized VMI is not a trivial task so non-experienced users would refrain from using EC3 if they



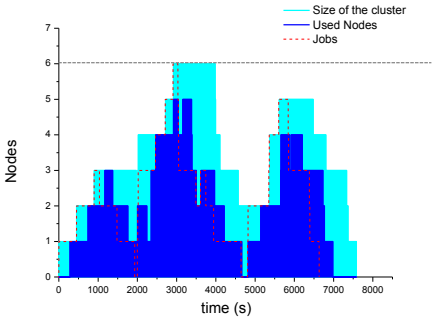
(a) Job pattern submission of the case study.



(b) Execution on VMs (EC3 with idle time for scale-in set to 30 min).



(c) Execution on containers (EC4Docker).



(d) Execution on VMs (EC3 with idle time for scale-in set to 10 min).

Figure 4: Execution results for both container-based cluster (a) and VM-based cluster (b) where *light blue* represents the number of virtual nodes deployed, *dark blue* depicts used nodes executing jobs and the *red* dashed line indicates the job pattern submission. The upper *grey* dotted line represents the limit size of the cluster, fixed to six nodes.

are required to prepare their own VMIs.

In contrast, creating a Docker container image from a Dockerfile is a much easier process than building a VMI. It is necessary to take into account that the container times shown in the table do not consider the time required to generate the container image from the Dockerfile, since this task only needs to be performed once by the administrator or the user. It is worth to point out that the time needed to create the container image is equivalent to the contextualization time employed by a non-preconfigured VM. Moreover, the time to pull the container images if they are stored in Docker Hub has not been included in the table, as this is performed only once, but it took an average of 150 s. in our tests.

	Scen. a)	Scen. b)		Scen. c)	
		Prec.	Non prec.	Prec.	Non prec.
Deployment avg. time	2	35	35	35	35
Active SSH avg. time	1	30	30	30	30
Total avg. time machine ready	3	65	65	65	65
FE contextualization avg. time	0	340	830	335	853
Total avg. time FE ready	3	405	895	400	918
WN contextualization avg. time	0	219	702	220	684
Total avg. time WN ready in LRMS	16	284	767	285	749
Job avg. waiting time	15.25	101	305	209	449
Job avg. execution time	994	1076	1083	1064	1128

Table 1: Time analysis, in seconds, for the different phases of the scenarios. Scenario a) refers to the container-based execution, scenario b) refers to the VM-based execution with an idle time configuration of 30 minutes and scenario c) refers to the VM-based execution with an idle time configuration of 10 minutes. In b) and c) the tests are carried out with preconfigured Virtual Machines Images (VMIs) (Prec.) and without preconfigured VMIs (Non prec.).

Second, we present in Figure 4 the results obtained from the execution of the job pattern submission show in Figure 4(a). Scenario a) is represented in Figure 4(c), scenario b) is shown in Figure 4(b)) and scenario c) is addressed in Figure 4(d). For the three executions, we have used conservative elasticity

policies to ensure the minimum costs for the infrastructure in terms of energy and resources consumption. Thus, CLUES has been configured to power on nodes according to the *1:1 start* strategy, i.e. when a job arrives to the LRMS and there is no available node to execute it, a virtual node is deployed. On the other hand, the power off policy selected was *delayed shutdown*, destroying nodes when they are idle for 2 minutes, for the scenario a) execution, 30 minutes for the scenario b) execution and 10 minutes for the scenario c). The differences in time for powering off a node are based on the time that a new virtual node needs to be ready for task execution (16 seconds in case of a container node and 285 seconds in average for a VM node). Scenarios b) and c) involve the same execution but the variations in the idle time to trigger the scale-in policy introduced differences in the behaviour of the cluster, as it can be appreciated in the figures.

Based on the results represented in Figure 4 we can highlight that the container-based cluster deployed in scenario a) fits almost perfectly to the workload of the computing cluster. Indeed, containers only take a few seconds to be ready to execute the jobs of the cluster since the contextualization process is not required, and starting a container is faster than booting a VM. Therefore, the average time that a job is queued up at the LRMS, i.e. in PENDING state, does not exceed 15 seconds.

In contrast, in scenarios b) and c) we can easily denote the differences deploying a node, that takes an average of 285 seconds to be ready and detected by the LRMS as an eligible node to execute jobs. This situation is represented in Figure 4(b),(d) in light blue, and covers the time needed to deploy a new VM, obtain SSH access to it and contextualize the job execution environment. For example, in Figure 4(b), for the first job this requires the initial 280 seconds of the execution. This situation is repeated for the subsequent jobs that arrive to the LRMS, when no available nodes are in the cluster. However, once the cluster is fully deployed, new jobs do not need to wait for additional nodes to be deployed. Instead, they just wait for other tasks to finish. This fact helps reducing the total average time of jobs waiting in the LRMS queue, which is

101 seconds. However, the resources are not properly exploited, because most of the nodes were idle a long period of time.

This situation can be better addressed by reducing the idle time allowed for nodes as it is done in scenario c). In this case, the available resources are better used, but the total time of execution increases (6989 seconds) like the job average waiting time (209 seconds) in contrast with the other two scenarios a) and b). However, despite the differences in the time required to provision new nodes in all scenarios, the total execution time in scenarios a) and b) is very similar. Container-based execution (scenario a)), requires 6579 seconds to complete all the submitted jobs while VM-based execution (scenario b)) takes 6661 seconds. Note that, on the one hand, scenario b) is significantly impacted by requiring to deploy additional nodes (VMs) at the beginning but the deployment and configuration of the nodes is produced concurrently. However, once the new nodes are up and running, jobs can be processed on a first-come-first-served basis. On the other hand, scenario c) is also impacted by the initial deployment of new VMs. However, the infrastructure does not maintain the nodes active and more time dedicated to deploy nodes os needed during the execution. These facts reveal that in a VM-based execution, increasing the time that nodes are idle, reduces the total execution time (no extra time is dedicated to deploy nodes to scale out) at the expense of wasting computational resources. Also, if the idle time to trigger a scale in operation is reduced, the total time of execution increases (due to the extra time required to provision additional nodes, which increases the job waiting time) but the computational resources are better used.

It is of special relevance the differences in the average time for a single job execution, that is an 8.2% faster in the containers deployed in the scenario (a) (994 seconds), than in VMs ([1064-1128] seconds). This fact confirms the higher overheads in CPU and memory that VMs suffer, comparing with the light-weight virtualization introduced by Docker containers.

Figure 4 does not represent the time required to deploy and configure the front-end of the cluster. This data is presented in Table 1, where for a container-based cluster this task only requires deploying a container in Docker Swarm and

requesting a redirection to IPFloater (3 seconds). Meanwhile, for a VM-based cluster, this task involves the creation of a new VM in OpenNebula, wait until the SSH of the VM is active and complete the contextualization process ([895-918] seconds in average for a non-preconfigured VMI and [335-340] seconds for a preconfigured VMI).

All the analyzed results suggest that containers are a proper solution to execute groups of short HTC (High Throughput Computing) tasks, like Bag of Tasks (BoT) applications. Indeed, for short tasks the required deployment time of a VM-based working node clearly outweighs the execution time of the tasks. HPC tasks can also benefit from the reduced overheads that arise when using containers. In contrast, for longer tasks, contextualization time may become negligible with respect to the total execution time and, therefore, these tasks can take advantage of the unlimited resources offered by Cloud Computing platforms in the shape of VMs.

4.2. Discussion

As it occurs in physical clusters, in order to use the virtual cluster it is recommended to introduce some other tools that enhance the features of the cluster and also take benefit from virtualization techniques. One of the most noticeable examples is the mechanisms that ensure the availability and the reliability of the cluster. One benefit of virtual clusters with respect to physical clusters is that virtualization facilitates the relocation of nodes. Indeed, incidents such as power outages or network failures can introduce a downtime for users of physical clusters. In the case of virtual clusters, any of the computing nodes (i.e. front-end or working nodes) can be hosted in another virtualization infrastructure, thus maintaining the service to users. Concerning high availability, this can be achieved in EC4Docker by deploying multiple containers configured to act as front-ends and to configure high availability middleware, such as a load balancer that supports failover.

It is important to point out that container-based elastic clusters improve the overall performance compared to VM-based elastic clusters. As demonstrated

by the case study, the reduced footprint of the container images with respect to the virtual machine images enhances the ability of the elastic cluster to cushion the workload peaks. Booting the container-based virtual working nodes takes significant less time than the VM-based ones. Therefore, the average waiting time for a job to be running is considerably reduced.

Regarding the performance of the scientific computing clusters, containers executed in one host take profit from the fact that the computational resources are not allocated to a specific container. Instead, the default behaviour for the containers is to share the available resources, managed by the host OS. That means that if one container is executed in an 8-core host, the application running in the container will be able to use the 8 cores and the whole memory if there are no other competing containers. However, a VM deployed with a fixed number of cores and memory, will only be able to use that number of cores and amount of memory even if the rest of the physical host is idle.

5. Conclusions and Future Work

This paper has analyzed the feasibility of using Docker containers to support the creation of virtual elastic computer clusters for the execution of scientific applications. These clusters maintain the very same interfaces for end users but benefit from the reduced overheads introduced by containers. For this, we introduced the open-source EC4Docker tool to support the deployment of such clusters on a Container Orchestration Platform managed by Docker Swarm.

We have demonstrated the feasibility of adopting containers to execute scientific applications, introducing two main advantages when compared to traditional VMs: i) the low deploying times for new working nodes, and ii) potential reductions in the overhead caused by VMs in CPU, memory and storage, offering near-native performance. Moreover, from the discussed case study, we can conclude that container-based virtual clusters are an appropriate solution for the execution of short HTC tasks.

Future work involves the automatization of the generation of the container

images that EC4Docker uses to deploy the cluster. Currently, the administrator or the users need to generate their own images including the Dockerfile provided with EC4Docker in order to deploy their own applications in the container cluster environment. A service will be implemented to facilitate this process for non-experienced users. Finally, a thorough scalability testing will be carried out to quantify the benefits of the container technology versus virtual machines for the processing of jobs on scientific computing virtual clusters.

Acknowledgement

This work has been developed under the support of the program "Ayudas para la contratación de personal investigador en formación de carácter predoctoral, programa VALi+d", grant number ACIF/2013/003, from the Conselleria d'Educació of the Generalitat Valenciana. The authors wish to thank the financial support received form The Spanish Ministry of Economy and Competitiveness to develop the project "CLUVIEM", with reference TIN2013-44390-R.

References

[1] F. Vella, R. M. Cefal, A. Costantini, O. Gervasi, C. Tanci, GPU Computing in EGI Environment Using a Cloud Approach, in: 2011 International Conference on Computational Science and Its Applications, IEEE, 2011, pp. 150–155. doi:10.1109/ICCSA.2011.61.
URL http://ieeexplore.ieee.org/document/5959549/

[2] S. Camarasu-Pop, T. Glatard, H. Benoit-Cattin, Simulating Application Workflows and Services Deployed on the European Grid Infrastructure, in: 2013 13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing, IEEE, 2013, pp. 18–25. doi:10.1109/CCGrid.2013.13.
URL http://ieeexplore.ieee.org/document/6546054/

[3] P. Mell, T. Grance, The NIST Definition of Cloud Computing. NIST Special Publication 800-145 (Final), Tech. rep. (2011).

URL http://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-145.pdf

[4] AdaptiveComputing, Torque resource manager, http://www.adaptivecomputing.com/products/open-source/torque/, [Online; accessed 12-January-2015].

[5] M. A. Jette, A. B. Yoo, M. Grondona, Slurm: Simple linux utility for resource management, in: In Lecture Notes in Computer Science: Proceedings of Job Scheduling Strategies for Parallel Processing (JSSPP) 2003, Springer-Verlag, 2002, pp. 44–60.

[6] D. Thain, T. Tannenbaum, M. Livny, Distributed computing in practice: the condor experience., Concurrency - Practice and Experience 17 (2-4) (2005) 323–356.

[7] S. U. Ahn, S. O. Park, J. Kim, Profiling Job Activities of Batch Systems in the Data Center, in: 2016 International Conference on Platform Technology and Service (PlatCon), IEEE, 2016, pp. 1–5. doi:10.1109/PlatCon.2016.7456818.
URL http://ieeexplore.ieee.org/document/7456818/

[8] M. Caballer, C. de Alfonso, F. Alvarruiz, G. Moltó, EC3: Elastic Cloud Computing Cluster, Journal of Computer and System Sciences 79 (2013) 1341–1351. doi:10.1016/j.jcss.2013.06.005.
URL http://authors.elsevier.com/sd/article/S0022000013001141

[9] A. Calatrava, E. Romero, G. Moltó, M. Caballer, J. M. Alonso, Self-managed cost-efficient virtual elastic clusters on hybrid Cloud infrastructures, Future Generation Computer Systems 61 (2016) 13–25. doi:10.1016/j.future.2016.01.018.
URL http://authors.elsevier.com/sd/article/S0167739X16300024http://linkinghub.elsevier.com/retrieve/pii/S0167739X16300024

[10] W. Felter, A. Ferreira, R. Rajamony, J. Rubio, An updated performance comparison of virtual machines and linux containers, in: Performance Analysis of Systems and Software (ISPASS), 2015 IEEE International Symposium on, 2015, pp. 171–172. doi:10.1109/ISPASS.2015.7095802.

[11] M. Scheepers, Virtualization and containerization of application infrastructure: A comparison, Vol. 21, University of Twente, 2014.
URL http://referaat.cs.utwente.nl/conference/21/paper/7449/virtualization-and-containerization-of-application-infrastructure-a-comparison.pdf

[12] A. Luzzardi, V. Vieux, Swarm: a docker-native clustering system., https://docs.docker.com/swarm/, [Online; accessed 7-March-2016].

[13] R. Buyya, High Performance Cluster Computing: Architectures and Systems, Prentice Hall PTR, Upper Saddle River, NJ, USA, 1999.

[14] C. Prabhu, GRID an Cluster Computing, PHI Learning, 2008.
URL https://books.google.es/books?id=evcgB7Qlix4C

[15] I. Melia, S. Puri, K. Owens, K. Thirumalai, S. Yellumahanti, L. Herrmann, M. Coggin, J. Fernandes, K. Craven, D. Juengst, Linux containers: Why theyre in your future and what has to happen first, Tech. rep., CISCO and Redhat (September 2014).

[16] C. Pahl, B. Lee, Containers and clusters for edge cloud architectures – a technology review, in: Future Internet of Things and Cloud (FiCloud), 2015 3rd International Conference on, 2015, pp. 379–386. doi:10.1109/FiCloud.2015.35.

[17] Linux containers, lxc, https://linuxcontainers.org/, [Online; accessed 18-January-2016].

[18] Canonical, LXD (2016).
URL https://linuxcontainers.org/lxd/introduction/

[19] CoreOS, rkt (2016).
URL https://coreos.com/rkt/

[20] Openvz, http://openvz.org/Main_Page, [Online; accessed 14-January-2016].

[21] S. Soltesz, H. Pötzl, M. E. Fiuczynski, A. Bavier, L. Peterson, Container-based operating system virtualization: A scalable, high-performance alternative to hypervisors, SIGOPS Oper. Syst. Rev. 41 (3) (2007) 275–287. doi:10.1145/1272998.1273025.
URL http://doi.acm.org/10.1145/1272998.1273025

[22] Docker, https://www.docker.com/, [Online; accessed 18-January-2016].

[23] R. Peinl, F. Holzschuher, F. Pfitzer, Docker cluster management for the cloud - survey results and own solution, Journal of Grid Computing (2016) 1–18doi:10.1007/s10723-016-9366-y.
URL http://dx.doi.org/10.1007/s10723-016-9366-y

[24] Kubernetes, http://kubernetes.io/, [Online; accessed 7-March-2016].

[25] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, I. Stoica, Mesos: A platform for fine-grained resource sharing in the data center, in: Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation, NSDI'11, USENIX Association, Berkeley, CA, USA, 2011, pp. 295–308.
URL http://dl.acm.org/citation.cfm?id=1972457.1972488

[26] Marathon, https://mesosphere.github.io/marathon/, [Online; accessed 19-January-2016].

[27] Chronos, https://mesos.github.io/chronos/, [Online; accessed 19-January-2016].

[28] N. Regola, J. C. Ducom, Recommendations for virtualization technologies in high performance computing, in: Cloud Computing Technology and

Science (CloudCom), 2010 IEEE Second International Conference on, 2010, pp. 409–416. doi:10.1109/CloudCom.2010.71.

[29] D. Bernstein, Containers and cloud: From lxc to docker to kubernetes, Cloud Computing, IEEE 1 (3) (2014) 81–84. doi:10.1109/MCC.2014.51.

[30] C. Zheng, 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, VTDC '15, ACM, New York, NY, USA, 2015, pp. 31–38. doi:10.1145/2755979.2755984.
URL http://doi.acm.org/10.1145/2755979.2755984

[31] W. Gerlach, W. Tang, K. Keegan, T. Harrison, A. Wilke, J. Bischof, M. D'Souza, S. Devoid, D. Murphy-Olson, N. Desai, F. Meyer, Skyport: Container-based execution environment management for multi-cloud scientific workflows, in: Proceedings of the 5th International Workshop on Data-Intensive Computing in the Clouds, DataCloud '14, IEEE Press, Piscataway, NJ, USA, 2014, pp. 25–32. doi:10.1109/DataCloud.2014.6.
URL http://dx.doi.org/10.1109/DataCloud.2014.6

[32] A. Slominski, V. Muthusamy, R. Khalaf, Building a multi-tenant cloud service from legacy code with docker containers., in: IC2E, IEEE, 2015, pp. 394–396.

[33] M. G. Xavier, M. V. Neves, F. D. Rossi, T. C. Ferreto, T. Lange, C. A. F. De Rose, Performance evaluation of container-based virtualization for high performance computing environments, in: Proceedings of the 2013 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, PDP '13, IEEE Computer Society, Washington, DC, USA, 2013, pp. 233–240. doi:10.1109/PDP.2013.41.
URL http://dx.doi.org/10.1109/PDP.2013.41

[34] J. Petazzoni, Linux containers(lxc), docker, and security, http://www.slideshare.net/jpetazzo/

linux-containers-lxc-docker-and-security/4-Fear_Uncertainty_and_DoubtLXC_is, [Online; accessed 18-January-2016].

[35] T. Adufu, J. Choi, Y. Kim, Is container-based technology a winner for high performance scientific applications?, in: Network Operations and Management Symposium (APNOMS), 2015 17th Asia-Pacific, 2015, pp. 507–510. doi:10.1109/APNOMS.2015.7275379.

[36] MIT, Starcluster, http://web.mit.edu/stardev/cluster/, [Online; accessed 1-December-2014].

[37] U. of Zurich, Elasticluster, http://gc3-uzh-ch.github.io/elasticluster/, [Online; accessed 21-January-2016].

[38] M. G. Xavier, M. V. Neves, C. A. F. de Rose, A performance comparison of container-based virtualization systems for mapreduce clusters, in: Parallel, Distributed and Network-Based Processing (PDP), 2014 22nd Euromicro International Conference on, 2014, pp. 299–306. doi:10.1109/PDP.2014.78.

[39] H. V. Higgins J., V. C., High Performance Computing: 30th International Conference, ISC High Performance 2015, Frankfurt, Germany, July 12-16, 2015, Proceedings, Springer International Publishing, Cham, 2015, Ch. Orchestrating Docker Containers in the HPC Environment, pp. 506–513. doi:10.1007/978-3-319-20119-1_36.
URL http://dx.doi.org/10.1007/978-3-319-20119-1_36

[40] J.-W. Park, J. Hahm, Container-based cluster management platform for distributed computing, in: Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications, 2015, pp. 34–40.

[41] F. Alvarruiz, C. de Alfonso, M. Caballer, V. Hernández, An energy manager for high performance computer clusters, in: Proceedings of the 2012 IEEE 10th International Symposium on Parallel and Distributed Processing with

Applications, ISPA '12, IEEE Computer Society, Washington, DC, USA, 2012, pp. 231–238. doi:10.1109/ISPA.2012.38.
URL http://dx.doi.org/10.1109/ISPA.2012.38

[42] J. Duato, A. J. Pena, F. Silla, R. Mayo, E. S. Quintana-Orti, rCUDA: Reducing the number of GPU-based accelerators in high performance clusters, in: 2010 International Conference on High Performance Computing & Simulation, IEEE, 2010, pp. 224–231. doi:10.1109/HPCS.2010.5547126.
URL http://ieeexplore.ieee.org/document/5547126/

[43] Openstack, Openstack floating ips, http://docs.openstack.org/openstack-ops/content/floating_ips.html, [Online; accessed 21-May-2016].

[44] J. Goecks, A. Nekrutenko, J. Taylor, T. G. Team., Galaxy: a comprehensive approach for supporting accessible, reproducible, and transparent computational research in the life sciences., Genome biology 11 (8) (2010) R86.

[45] Mrbayes: Bayesian inference of phylogeny, http://mrbayes.sourceforge.net/index.php, [Online; accessed 22-April-2016].

[46] M. Software, Cynmix dataset, http://mrbayes.sourceforge.net/wiki/index.php/Cynmix.nex, [Online; accessed 05-Jun-2016].