2019 IEEE/ACM Workshop on Containers and New Orchestration Paradigms for Isolated Environments in HPC (CANOPIE-HPC)

Evaluation and Benchmarking of Singularity MPI Containers on EU Research e-Infrastructures

Víctor Sande Veiga*, Manuel Simon†, Abdulrahman Azab‡, Giuseppa Muscianisi§, Carlos Fernandez¶, Giuseppe Fiameni§ and Simone Marocchi§
*CIMNE - International Centre for Numerical Methods in Engineering, Spain
†Nokia, France
‡Division of Research Computing, University Center for Information Technology, University of Oslo, Norway
§CINECA - Interuniversity Consortium, Italy
¶CESGA - Centro de Supercomputación de Galicia, Spain
Email: *vsande@cimne.upc.edu, †eu@manuel.gal, ‡azab@usit.uio.no,
§{g.muscianisi,g.fiameni,s.marocchi}@cineca.it, ¶carlosf@cesga.es

***Abstract*—Linux Containers with the build-once run-anywhere principle have gained huge attention in the research community where portability and reproducibility are key concerns. Unlike virtual machines (VMs), containers run the underlying host OS kernel. The container filesystem can include all necessary non-default prerequisites to run the container application at unaltered performance. For this reason, containers are popular in HPC for use with parallel/MPI applications. Some use cases include also abstraction layers, e.g. MPI applications require matching of MPI version between the host and the container, and/or GPU applications require the underlying GPU drivers to be installed within the container filesystem. In short, containers can only abstract what is above the OS kernel, not below. Consequently, portability is not completely granted.**

This paper presents the experience of PRACE (Partnership for Advanced Computer in Europe) in supporting Singularity containers on HPC clusters and provides notes about possible approaches for deploying MPI applications using different use cases. Performance comparison between bare metal and container executions are also provided, showing a negligible overhead in the container execution.

I. Introduction

Sharing of software packages is an essential demand among scientists and researchers in order to reproduce results [1]. HPC centres are struggling to keep up with the rapid expansion of software tools and libraries. In some cases, large communities are developing software to serve their specific scientific community. In many cases, users are interested in tools that are difficult to install, due to long list of non-portable dependencies. Some requested software might be specifically targeted at an OS environment that is common for their domain but may conflict with the requirements from another community. For example, the biology and genomics community adopted Ubuntu as their base OS with a specific versions of Perl and Python [2].

Traditional software deployment performed in HPC presents some issues which make it very inflexible. Different approaches can provide an interesting alternative that is more flexible with very little impact on performance. Traditional software deployment and integration consists in performing all the needed steps, like download, configure, build, test and install, to have the software project natively in a production infrastructure. The main goal of the lightweight virtualization service in PRACE [3] is to provide a platform and best practices on how to offer various software with good performance and ready to use for end users. Scientific software is extremely complex from an architectural point of view. It is usually composed of numerous mathematical concepts and features implemented along several software components in order to provide high level abstraction layers. These layers can be implemented in the software project itself or integrated via third party libraries or software components. The whole environment of each scientific software is usually composed of a complex dependency matrix and, at least, one programming language and compiler. Isolation and integration of all dependency matrices at HPC clusters are traditionally managed by environment modules. Environment Modules provide a way to dynamically change the user environment through module files. The key advantage of environment modules is that it allows to use multiple versions of a program or package from the same account by just loading the proper module file. In general, module files are created on per application per version basis. They can be dynamically loaded, unloaded, or switched. Along with the capability of using multiple versions of the same software it also can be used to implement site policies regarding the access and use of applications. Module files allow managing the loading and unloading of environments to many particular applications, but to manage complex work-flows with environment modules can be sometimes non-affordable, and requires the re-installation of some tools with compatible dependencies. These issues are difficult to manage from the user and the administrator point of view. Finally, the hardware and software ecosystem of an HPC production infrastructure is different than a development ecosystem, and usually a lot of unexpected issues appear while integrating and deploying the complex environment of mathematical frameworks. To integrate the whole environment of each software project in a production infrastructure is usually hard and time consuming. Also, as this software is evolving very fast and using the latest technologies and features of the compilers, new versions are provided very frequently. This puts a lot of pressure on

978-1-7281-6028-3/19/$31.00 ©2019 IEEE
DOI 10.1109/CANOPIE-HPC49598.2019.00006

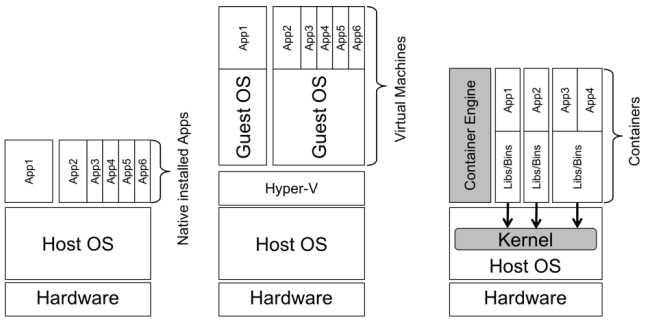

Fig. 1: Native vs Virtual Machines vs Containers

the software support team of the infrastructures. Even though there are some tools that could help in this process, like EasyBuild [4], an application to manage software on High Performance Computing (HPC) systems in an efficient way. What is intended to do is to find a simple and efficient way that allow to deploy and integrate that complex software with its dependencies. The goal is to reduce at maximum at possible the previously named time consuming. Most importantly, because end users can come up with endless combinations of software and dependencies, it is to achieve maximum portability, but without losing the computational capacity of the system.

This paper presents a research study proposing general approaches and configuration for running singularity MPI containers in an HPC environment. The aim is to obtain portability and flexibility for this container solution while running parallel applications, its configuration in the HPC infrastructure and the realization of multiple performance tests to ensure the usage of these HPC resources. The performed experiments and benchmarks have taken place on the MARCONI HPC cluster in CINECA, Italy [5], and Finis Terrae II HPC cluster in CESGA, Spain [6]

II. CONTAINERISATION

Linux "containerization" is an operating system level virtualization technology that offers lightweight virtualization. An application that runs as a container has its own root file-system, but shares kernel with the host operating system. Containers have many advantages over virtual machines (VMs). First, containers are less resource consuming since there no guest OS. Second, a container process is visible to the host operating system, giving the opportunity to system administrators for monitoring and controlling the behaviour of container processes. Linux containers are monitored and managed by a container engine which is responsible for initiating, managing, and allocating containers. Figure 1 depicts structural comparison between native deployment, VMs, and Linux containers.

Although the containerization techniques is a buzzword nowadays especially in the Datacenter and Cloud industry, the idea is quite old. Container or "chroot" (change root) was a Linux technology to isolate single processes from each other without the need of emulating different hardware for them. Containers are lightweight operating systems within the Host Operating system that runs them. It uses native instructions on the core CPU, without the requirement of any VMM (Virtual Machine Manager). The only limitation is that we have to use the Host operating systems kernel to use the existing hardware components, unlike with virtualization, where we could use different operating systems without any restriction at the cost of the performance overhead. This is the main target for this work. We can use different software, libraries and even different Linux distributions without reinstalling the system. This makes HPC systems more flexible and easy to use for scientists and developers. Container technology has become very popular as it makes application deployment very easy and efficient. As people move from virtualization to container technology, many enterprises have adopted software container for cloud application deployment.

A. HPC container platforms

There is a stiff competition to push different technologies in the market. To choose the right technology, depending on the purpose, it is important to understand what each of them stands for and does.

1) Docker [7]: Is the most popular container platform, and is the standard for the micro-service deployment model. Docker is more suitable for service-like processes rather than job-like ones and is not designed for running HPC distributed applications. The platform relies on the Docker engine and daemon to be installed and running on each node. Each container has its own process which is fired up by the docker daemon. Containers run by default as root. While this can be avoided by using user namespaces, getting access to the main `docker` command, users can easily escape this limitation, which introduces a security risk. Further more, each docker engine has its own image repository that, so far, cannot be shared with other docker engines. This means that running a parallel application X on e.g. 100 nodes, a X docker image need to be replicated 100 times, one per node. Docker so far has no support for MPI.

2) Shifter [2]: is a platform to run Docker containers that is developed by the National Energy Research Scientific Computing Center (NERSC) and is deployed in production on a Slurm cluster of Cray supercomputers. Shifter uses its own image format to which both Docker images and VMs are converted. The Docker engine is replaced by *image manager* for managing the new formated images. Previously NERSC introduced MyDock [2] which is a wrapper for Docker that enforces accessing containers as the user. MyDock however did not provide a solution for enforcing the inclusion of a running container in the cgroups associated with the Slurm job. In addition, both shifter and myDock enforces accessing as the user by first running as root and mounting `/etc/passwd` and `/etc/group` in each single container, then lets the user access the container as him/herself. Despite the fact that Shifter has proven to be useful managing HPC workflows, to use images in shifter, usesers must submit their images to a root controlled *image gateway* via a RESTful API. In addition, shifter is not currently well maintained.

3) Charliecloud [8]: is an open source project which is based on a user-defined software stack (UDSS). It is designed

to manage and run docker based containers, and One important advantage that it is very lightweight. It enables running containers as the user, by implementing user namespaces. The main disadvantage is the requirement of a recent kernel (4.4 is recommended) that supports user namespaces. Many EU HPC clusters still have their compute nodes with old kernels (< 3.10), i.e. don't support user namespaces which is a showstopper for charliecloud.

4) Podman [9]: is developed and maintained by Red Hat. It is a lightweight runtime for managing and running docker containers without the overhead of the docker engine and daemon. Podman though is not mainly designed for and is currently lacking support for HPC workloads. Podman can run rootless, but to to that (similar to charliecloud), it uses user namespaces (which is a limitation as stated in Section II-A3). In addition, its rootless mode is not compatible with most parallel filesystems.

5) Singularity [10]: Singularity is the third and last HPC-specific container platform (after shifter and charliecloud). It is developed and maintained by Sylabs Inc. Unlike shifter, it doesn't have that complex architecture (with image manager and gateway) and can be simply installed as a environment module. Unlike Podman and charliecloud, it does not require kernel user namespaces to run rootless containers, and thus can be deployed on old Linux kernels. Singularity supports both Docker image format and its own native Single Image "flat" Format (SIF). Singularity is the most widely deployed container platform on HPC systems, currently there are 25,000+ systems running Singularity.

6) uDocker [11]: is a basic user tool to execute simple Docker containers in user space without requiring root privileges, which enables basic download and sequential execution of docker containers by non-privileged users in Linux systems. It can be used to access and execute the content of docker containers in Linux batch systems and interactive clusters that are managed by other entities such as grid infrastructures or externally managed batch or interactive systems. uDocker and Singularity were developed specifically to be used in HPC environments, as we will describe below. Both of them are Docker-compatible, and help to empower end-users of HPC systems providing a contained location where to manage their installations and custom software. They are also a great solution for developers, one of the biggest benefits for them is to deliver their software in a controlled environment and ready-to-use. Although the uDocker development team is working to integrate it with message passing interface libraries (MPI), unfortunately, it is not yet maturely supported. uDocker is more of a wrapper than a standalone platform.

B. Container platform choice

Several containerization technologies (like LXC [12], Docker, Udocker [11] and Singularity [13]) have been tested in the context of this study, but finally, we decided to go with singularity. Docker was rejected because of its kernel requirements and security. Podman and charliecloud were rejected for their kernel requirements. Finis Terrae II and many other EU HPC platforms do not have recent kernels

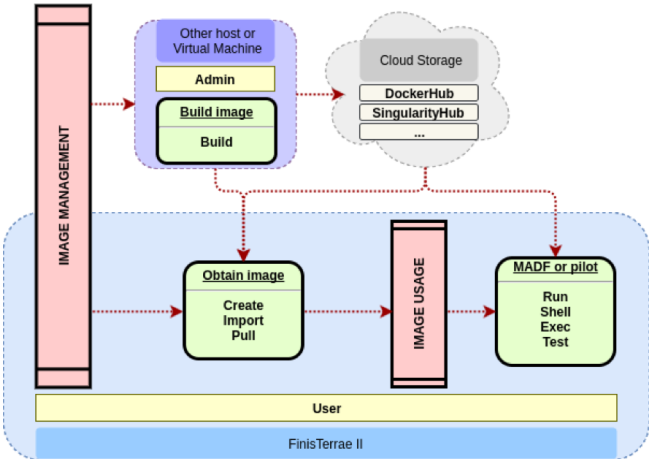

Fig. 2: Singularity work-flow at Finis Terrae II

on compute nodes. uDocker was rejected for the lack of maturity for its support for MPI. Shifter was rejected due to the many different components which are needed to work together in a complex architecture and also for the lack of maintenance. Singularity was designed focusing on HPC and allows to leverage the resources of whatever host in which the software is running. This includes HPC interconnects, resource managers, file systems, GPUs and/or accelerators, etc. Singularity was also designed around the notion of extreme mobility of computing and reproducible science. Singularity is also used to perform HPC in the cloud on AWS, Google Cloud, Azure and other cloud providers. This makes it possible to develop a research work-flow on a laboratory or a laboratory server, then bundle it to run on a departmental cluster, on a leadership class supercomputer, or in the cloud. The simple usage of Singularity allows users to manage the creation of new containers and also to run parallel applications easily. Figure 2 describes the installation in FinisTerrae II, users can pull images or execute containers in FT2 from public registries, and also import images from tar pipes. Once the image is created, Singularity allows executing the container in interactive mode, and test or running any contained application using batch systems. All the work-flow can be managed by a normal user at FinisTerrae II, except the build process that needs to be called by a superuser. We can use a virtual machine with superuser privileges to modify or adapt an image to the infrastructure using the Singularity build command.

We can use a virtual machine with superuser privileges to modify or adapt and image to the infrastructure using the bootstrap Singularity command. This work-flow allow an automatized integration and deployment, deriving in a very flexible and portable solution.

III. Containerization of Parallel MPI Applications

Container technologies can be used to handle portability issues. The different environments inside the containers can coexist on the same machine and share the OS kernel, each running as isolated processes in user space. In what follows, a

review about the MPI libraries and network driver installation within a Singularity container is provided. Performance test are presented, as a performance comparison among bare metal and container execution of Quantum Espresso code [14], showing a negligible difference in the total execution time growing the number of cores used and using different MPI libraries and interconnection network. To do this, it is needed first of all the availability of a Message Passing Interface (MPI) library inside the container. Moreover, since a HPC system is usually characterized by having a high-bandwidth low-latency interconnection among nodes, leveraging Remote Direct Memory Access (RDMA), the application has to be aware of such hardware. So, particular drivers and libraries should be available in the container, enable it to leverage such network. In this section, informations about the MPI utilization within a container environment are provided, focusing on Open MPI [15] and Intel MPI [16]. The OFED libraries have been available in the container, in order to make it able to run over an Infiniband network. The particular case of Intel Omni-Path Architecture is considered.

A. Building notes of a Singularity container

Applications suitable for HPC system use generally parallel libraries, taking advantage from high-bandwidth and low-latency interconnection among nodes. To execute a parallel MPI application within a Singularity container, MPI library has to be available both in the host cluster and in the container [17], [13]. The availability of the MPI library in the container can be achieved installing it from scratch or binding it from those available in the host system. Standard implementations of MPI as Open MPI [15] or Intel MPI [16] can be used, but also less common choices are possible. Attention has to be paid to possible compatibility issues among different MPI version in the host and in the container [18]. To build containers able to connect to networking fabric libraries in the host, some additional libraries have to be installed. As an example, if the cluster has an infiniband network, the OpenFabrics Enterprise Distribution (OFED) libraries have to be available into the container. An other technology available for HPC cluster is the Intel Omni-Path Architecture (OPA), that offers high performance networking interconnect technology with low communications latency and high bandwidth characteristics ideally suited for HPC applications. HPC applications run in containers can take advantage of the improved network performance of the Intel OPA technology [19], also.

IV. Singularity MPI Integration

Singularity relies on other libraries (MPI, PMI) to provide the MPI hybrid approach. To increase MPI containers portability, the work done from PMIx (OpenPMIx) team [18] is definitely an important improvement. The hybrid approach requires that MPI must exist both inside and outside the container. The Open MPI/Singularity workflow invocation pathway is as follows:

1) From shell (or resource manager) mpirun gets called
2) mpirun forks and exec orte daemon
3) Orted process creates PMI
4) Orted forks == to the number of process per node requested
5) Orted children exec to original command passed tompirun (Singularity)
6) Each Singularity execs the command passed inside the given container
7) Each MPI program links in the dynamic Open MPI libraries (ldd)
8) OpenMPI libraries continue to open the non-ldd shared libraries (dlopen)
9) OpenMPI libraries connect back to original orted via PMI
10) All non-shared memory communication occurs through the PMI and then to local interfaces (e.g. InfiniBand)

This entire process happens behind the scenes, and from the user's perspective running via MPI is as simple as just calling mpirun on the host as they would normally, but there are some important considerations to integrate Singularity and OpenMPI in an HPC environment:

- OpenMPI must be newer of equal to the version inside the container.
- To support InfiniBand, the container must support it.
- To support PMI, the container must support it.
- Very little (if any) performance penalty has been observed.

To achieve proper containerized OpenMPI support, OpenMPI version 2.1 should be used. However, Singularity team explain that there are three caveats:

1) OpenMPI 1.10.x may work but we expect you will need exactly matching version of PMI and Open MPI on both host and container (the 2.1 series should relax this requirement).
2) OpenMPI 2.1.0 has a bug affecting compilation of libraries for some interfaces (particularly Mellanox interfaces using libmxm are known to fail). If your in this situation you should use the master branch of Open MPI rather than the release.
3) Using Open MPI 2.1 does not magically allow your container to connect to networking fabric libraries in the host. If your cluster has, for example, an infiniband network you still need to install OFED libraries into the container. Alternatively you could bind mount both Open MPI and networking libraries into the container, but this could run afoul of glib compatibility issues (its generally OK if the container glibc is more recent than the host, but not the other way around).

V. Different approaches for deploying MPI containers

Singularity does not magically allow containers to connect to networking fabric libraries in the host. As the cluster makes use of an InfiniBand network, it will be needed to install OFED libraries into the container. So, all the needed libraries will be installed during the creation process of the containers. That is, it is important to specify all this software downloading, installation and configuration into the bootstrap configuration file. For a clearer and more concise language, from now on,

Container OpenMPI	Host OpenMPI				
	2.0.0	2.0.1	2.0.3	2.1.1	3.0.0
2.0.0	☑	☒	☒	☒	☑
2.0.1	☒	☑	☒	☒	☑
2.0.3	☒	☒	☑	☒	☑
2.1.1	☒	☒	☒	☑	☑
3.0.0	☑	☑	☑	☑	☑

Fig. 3: Mixing different mpirun compilers [20]

whenever we refer to containers with Open MPI inside, we will do it with the following nomenclature: container ompi/Open MPI version. E.g. if we are talking about a Ubuntu 16.04.2 LTS (Ubuntu Xenial) container with Open MPI version 1.10.2, we will just call it: container ompi/1.10.2. In case of Intel MPI, the nomenclature will be changed to container impi/Intel MPI version (e.g. container impi/2017). That from the container side. Changing to the outside values, we already know that the cluster is a Red Hat Enterprise Linux Server release 6.7, which have different Open MPI and Intel MPI versions installed. So, every time we want to talk about an external configuration that make use of a concrete version of Open MPI or Intel MPI we will call it host ompi/version or host impi/version. Then, different approaches are needed in order to arrive at a solution that allows the use of different versions of MPI libraries inside and outside the container. We will focus our attention on the ompi configurations, as it is a more standardized configuration, but some tests with impi configurations will be done too.

A. Mixing MPI versions

The first approach was the more intuitive and expected. That is, try a simple application that make use of MPI, mixing the Open MPI inside and outside the container. For this first approach, the selected process manager was mpirun. The test done was the mix of the different ompi possibilities, including the different available compilers. For these executions, the obtained results, shown in Figure 3 were a one-to-one version compatibility, except for 3.0.0. That is, for the properly program execution, the version of OpenMPI must match exactly inside and outside the container for Open MPI less than 3.0.0

B. Install the same OpenMPI version

This is the safest approach, and under it we must match exactly the same Open MPI version at the host and within the container. So, if we want to take this approach to production, OpenMPI versionmust match exactly within the container and the host.

C. MPI Binding

This approach was made under the srun process manager. A exhaustive study has been done to detect all the OpenMPI dependencies, all of themlocated at `/lib64/` and `/usr/lib64/` at the Finis Terrae II. So, once dependencies were obtained, we observed them carefully. Then, making use of the Singularity binding option, we bind the needed host libraries, forcing the containers to use these selected libraries instead of their owns. It is quite important to note that simple binding and replacing the entire directories inside the container is not the right solution as it supposes a replacement of the whole low-level library and kernel modules. We need to create an extra level of indirection with symbolic links to get control on which libraries will be replaced inside the container. An extra preparation is needed in this approach. We need to create the folders where the bindings will be done, because Singularity will not automatic create them. The binding folders preparation starts with the creation of a folder hierarchy where there are included the folders where the bindings will be done as well as folders which will contain exactly the same content as the host important libraries.

Looking at the folders hierarchy, we have copied the original host folders: `/lib64`, `/mpi/lib` and `/usr/lib64`. The reason why there are two folders inside the `/mpi/lib` directory (ompi_1.X and ompi_2.X) it is because several changes happen when the Open MPI version changed from 1.X to 2.X. One prove of these several changes is reflected in the backward compatibility loss in the Application Binary Interface (ABI). The loss of compatibility can be tracked making use of the ABI tracker tool [21]. Then, when a important dependency is found, the symbolic link is copied to the parent folder, that is where the binding is done. As this approach could result a little confusing to the reader, a graphic representation was done in order to facilitate the understanding of it. This graphic representation can be observed on Figure 4. We have two important blocks: the host and the container, being this last one hosted by the first one in some folder. Both of them are Linux-based systems, so they have the typically Linux folder hierarchy. In the host, we can observe an important folder called ”filtered”, where we had carefully created those symbolic links to the libraries that we will use to do indirections. Turning into the container, there is an important folder called `/host/`, which is no more than the binding of the previously called host folders. Inside it, we can observe the filtered folder, where now the symbolic links points to existing files. Those symbolic links are represented by discontinuous lines, while the bindings are represented by the continuous ones. Explaining the Figure 4 in a more structured way, this is what we do:

1) Bind the OpenMPI libraries and their dependencies located at `/usr` and `/lib64` (continuous lines).
2) Create directories with symbolic links to the selected libraries. These links will be properly solved inside the container (the ”filtered” folder).
3) Bind directories with symbolic links (discontinuous lines).
4) Export `LD_LIBRARY_PATH` pre-pending the binded Open MPI libraries, dependencies and directories with symbolic links.

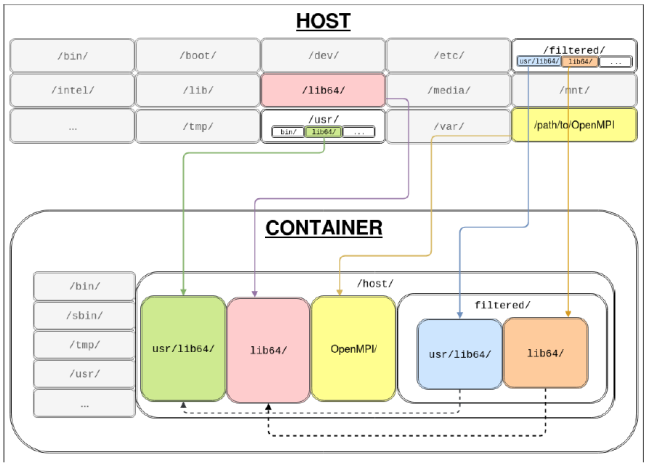

Fig. 4: Binding approach: graphic representation

D. Discussion

Mixing MPI versions (container and host) produces compatibility issues and is no go for Open MPI less than 3.0.0. To support MPI containers there are two options: Containers can be built with all the needed library inside or they can bind the library of the host, both options have pros and cons, and there is no best option to recommend. The binding allow to create smaller container in size, but the host's library and their paths have to be able to interface with the binary in the container, this is called application binary interface (ABI) compatibility. The ABI compatibility is not always possible and a specific binding pattern reduce the general portability of the container on other servers. Putting the library inside the container produce larger container but also in that cases, putting hardware specific library (OFED) can introduce compatibility issues when the host hardware change. The performances of both approaches are equivalent. However neither of them can be used as black box, the knowledge of the container and host structure is always needed. Here the pros and cons of each of the later approaches are discussed.

1) Install the same OpenMPI version: Pros:

- Containers' philosophy is not corrupted.
- We can ensure 100% ABI compatibility.

Cons:

- High system administrator iteration. The system administrator is responsible for installing every needed version of Open MPI at the host, as well as installing and configuring each new version that is released.
- We need to inspect every single container in order to know which Open MPI version it has installed inside. Once this is known, we need to deploy the exactly match version at the host.
- Container must have installed some libraries in order to use some HPC resources (libiverbs, ibutils, etc.)

2) MPI binding: Pros:

- Medium system administrator iteration. The system administrator must install and configure at least one Open MPI version per major version available.
- We can use the srun process manager as we are using exactly the same PMI.
- High interoperability level.
- The container does not need to have installed specific hardware related libraries.

Cons:

- Containers' philosophy is corrupted. This do not damage the containers and they can be employed under different environments and hosts with no changes on them.
- In general, we can not ensure 100% ABI compatibility. Reader can read the ABI tracker page if he wants to know more.
- Similar as it happened with the previous approach, we need to inspect every single container in order to know which Open MPI major version it has installed inside.
- All containers must go through a bootstrap process to make their environment suitable for our configuration (create a series of folders, make links, etc.)

VI. Use cases and benchmarking results at MARCONI@CINECA

In CINECA, a "Install the same Open MPI version" approach was chosen, instead of "MPI binding".

1) System description: MARCONI [5] is the Tier-0 CINECA system based on Lenovo NeXtScale platform. The current configuration of MARCONI consists of:

- 3600 Intel Knights Landing nodes, each equipped with 1 Intel Xeon Phi 7250 @1.4 GHz, with 68 cores each and 96 GB of RAM, also named as MARCONI A2 - KNL
- 3216 Intel SkyLake nodes, each equipped with 2 Intel Xeon 8160 @ 2.1 GHz, with 24 cores each and 192 GB of RAM, also named as MARCONI A3 - SKL.

This supercomputer takes advantage of the Intel Omni-Path Architecture, which provides the level of high-performance inter-connectivity required to efficiently scale out the system's thousands of servers. A high-performance Lenovo GSS storage subsystem, that integrates the IBM Spectrum Scale™ (GPFS) file system, is connected to the Intel Omni-Path Fabric and provides data storage capacity for about 10 PBytes net. MARCONI A3, SkyLake node partitions, was used for testing. On those nodes, the operating system is CentOS 7.3.1611. The software is available through environment modules, and the Singularity version used for testing is 3.0.1. A compatible Intel OPA driver was installed in the containers. All runs have been submitted to MARCONI A3 as Slurm jobs and all the nodes have been used in exclusive way, without competition with other runs.

2) Test case 1: containerized versus bare metal execution using Intel MPI library and Intel OPA interconnection network: A Singularity container has been built by installing Intel MPI library and all necessary drivers for using the Intel OPA interconnection network available in MARCONI. The version of Quantum Espresso code used is 6.3, compiled with Intel parallel studio 2018 - update 4. The container has been built using Singularity 3.0.1 . As a test, a slab of Zirconium Silicide material has been used, a mineral consisting of zirconium, and

silicon atoms. This slab consists of 108 atoms in total with a K-point mesh of $6 \times 6 \times 5$.

On the basis of the performance suggestions available in QE documentation [22], hybrid MPI+OpenMP jobs have been submitted, because it has been demonstrated a performance improvements respect to a pure MPI job when a large number of nodes (>64) are used. During the compilation phase of Quantum Espresso, Intel Scalapack library (part of Intel Math Kernel Library (MKL) [23]) has been included, to maintain performances for a large number of nodes. In the proposed tests, the QE parameters were set to 8 OMP Threads, a number of MPI tasks equal to the number of cores divided by the number of OMP Threads, "npool" parameter equal to 16, and "ndiag" parameter equal to the number of MPI Tasks divided by the parameter "npool". Figure 5 shows a comparison between the performance in terms of execution time for bare metal vs container with increasing the number of cores up to 6144 for the input provided (i.e. up to 128 MARCONI Sky Lake nodes). The total execution time is reported for the entire code (PWSCF) and for the Fastest Fourier Transform in the West (FFTW), which is one of the most time-consuming codes. Each computation has been repeated 10 times, and the displayed values in the figures include: minimum, median, and maximum. The calculations were repeated 5 times and then averaged on the set of data. Figure 5 shows a negligible difference in the total execution time of bare metal and container up to 6144 cores (128 nodes).

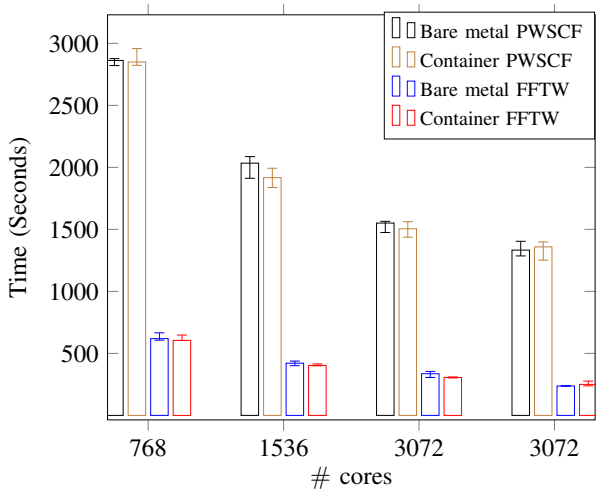

Fig. 5: Total execution time of the simulation (PWSCF) and Fastest Fourier Transform in the West (FFTW) are plotted for a system of 108 zirconium and silicon atoms running on 16, 32, 64 and 128 MARCONI Sky Lake nodes, every node has 48 cores

3) Test case 2: containerized versus bare metal execution using Open MPI and Intel MPI: In the second experiment the performances of two different compilers has been considered, Intel MPI and Open MPI. In both cases, the Quantum Espresso code is used, simulting a 24 Zirconium and Silicon atoms, with a K-point mesh of $6 \times 19 \times 13$, pure MPI. A Singularity container with Open MPI library 2.1.1 and Quantum Espresso version 6.4 has been built, together with those already explained in Section VI-2. The Results are shown in Figure 6. The total execution time is reported for the entire code (PWSCF) and for the Fastest Fourier Transform in the West (FFTW). Each computation has been repeated 10 times, and the displayed values in the figures include: minimum, median, and maximum. The overhead introduced by containerized execution compared to bare metal is negligible. By comparing the performances of the two versions of Quantum Espresso compiled with two different MPI libraries Figure 6, it is noted that QE compiled with Intel MPI are about 5 times faster than compiled with Open MPI. This is expected and is not affected by the containerization procedure. Our goal performing these experiments was to test and prove the negligible effect of containerization on the execution time of for Quantum Espresso on MARCONI cluster both increasing the number of nodes and changing the compiler. Moreover, we were interested in testing that the scaling shape of QE is similar when using containers, independently from the compiler.

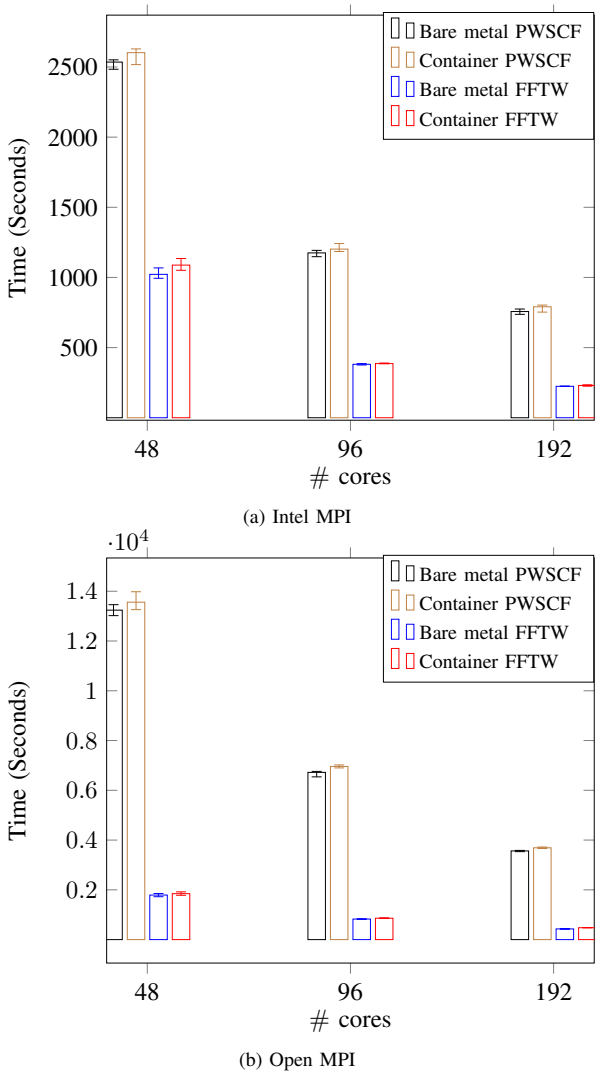

Fig. 6: Total execution time of PWSCF and Fastest Fourier Transform in the West (FFTW) on a system of 24 zirconium and silicon atoms run on 1, 2, 4 MARCONI Sky Lake nodes, where every node has 48 cores

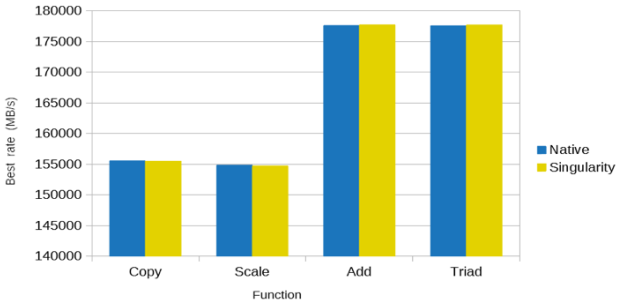

Fig. 7: STREAM best bandwidth rates comparison

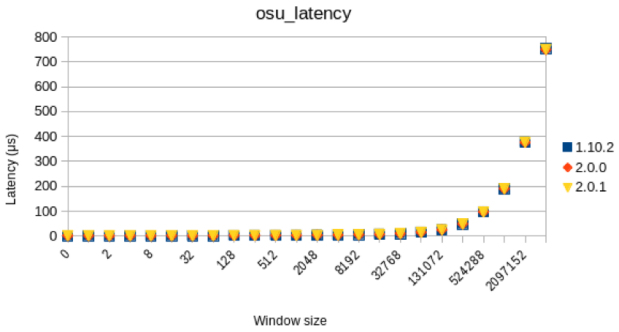

Fig. 8: Latency from Singularity using OSU micro-benchmarks

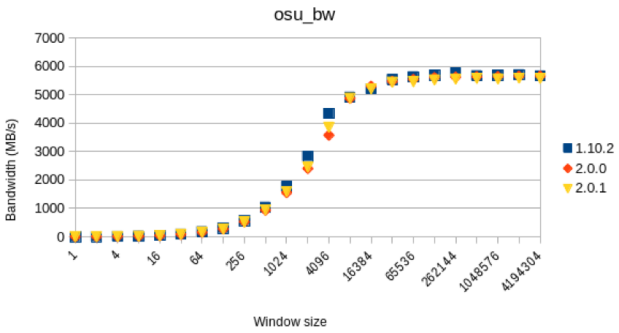

Fig. 9: Bandwidth from Singularity using OSU micro-benchmarks

VII. Use cases and benchmarking results at FinisTerrae II @ CESGA

Here we describe the use cases at FinisTerrae II.

The benchmarks were performed in order to demonstrate that Singularity is able to take advantage of the HPC resources, in particular Infiniband networks and RAM memory. For these benchmarks we used a base Singularity image with an Ubuntu 16.04 (Xenial) OS and several OpenMPI versions. For these benchmarks we took into account the MPI cross-version compatibility issue exposed in the previous section. The STREAM benchmark is de facto industry standards for measuring sustained RAM memory bandwidth and the corresponding computation rate for simple vector kernels. The MPI version of STREAM is able to measure the employed RAM under a multi node environment. The fact of using several nodes with exactly the same configuration helps us to check results consistency. In this case, two FinisTerrae II nodes, 48 cores, were utilized for running 10 repetitions of this benchmark natively and within a Singularity container with a global array size of 7.6×10^8, which is a big enough size to not be cacheable.

As we can see in the above figure, obtained bandwidth rates are really close between the native execution and the execution performed from a Singularity container, differences are negligible. Infiniband networks also have decisive impact on parallel applications performance and we have also benchmarked it from Singularity containers. We used the base Singularity container with three different OpenMPI versions (1.10.2, 2.0.0 and 2.0.1) together with OSU micro-benchmarks. OSU are suite of synthetic standard tests for Infiniband networks developed by MVAPICH. In particular, among the bunch of tests included we have performed those related with point-to-point communications in order to get results about typical properties like latency and bandwidth. Only two cores in different nodes were used for this benchmark. Latency tests are carried out in a ping-pong fashion. Many iterations of message sending and receiving cycles were performed modifying the size of the interchanged messages (window size) and the OpenMPI version used.

We can see in Figure 8 and Figure 9, unidirectional latency measurements are strongly related to the message size. For window sizes up to 8192 bytes we obtain less than 6 microseconds of latency, which are correct values for Infiniband networks. In this case the OpenMPI version does not have influence on the results. For the measurement of the bandwidth, we increase the windows size to saturate the network interfaces in order to obtain the best sustained bandwidth rates. In the figure below, we can observe that the general behaviour is as expected. The maximum bandwidth reached is close to 6GB/s, which are again in a correct value ranges for Infiniband. Although getting slightly different values depending on the OpenMPI version, we obtain similar results with critical values. From these benchmark results, we can conclude that Singularity containers running parallel applications are taking advantage of these HPC resources under the specified conditions.

Listing 1 presents one of two sbatch scripts responsible for reserving resources in the cluster. Keep in mind that these must adapt it according to the container name (right now depends on a variable, but it did not tend to be so). We must also adapt the number of processors, the maximum execution time, as well as the queue under we want it to run (thinnodes, cola-corta, software, etc.)

Listing 2 describes the template used to build the basic MPI Singularity container.

VIII. Related work

There are numerous efforts in the direction of performance benchmarking for container runtimes and containerised applications [24], [25], [26], [27], [28], [29], [30], [31], [32], [33]. IBM research produced a report [24] comparing the

Listing 1: SBATCH script for reserving resources on the cluster

```
#!/ bin/sh

# SBATCH -n 2
# SBATCH -N 2
# SBATCH -p thinnodes
# SBATCH -t 00:10:30

if srun singularity exec \
-B /lib64:/host/lib64 \
-B /usr/lib64:/host/usr/lib64 \
-B $HOST_LIBS/lib64:/host/filtered/lib64 \
-B $HOST_LIBS/usr/lib64:/host/filtered/usr/lib64 \
-B /opt/cesga/openmpi/$OPEN_MPI_VERSION/$COMPILER/
$COMPILER_VERSION/lib:/. singularity .d/ libs \
-B $HOST_LIBS/mpi:/host/mpi \
-B /opt/cesga:/opt/cesga \
-B /usr/lib64/slurm:/usr/lib64/slurm \
-B /etc/slurm:/etc/slurm \
-B /var/run/munge:/run/munge \
-B /etc/libibverbs.d:/etc/libibverbs.d \
-B /mnt:/mnt \
ubuntu ." $CONTAINER_OPEN_MPI_VERSION ".img bash -c "
export LD_LIBRARY_PATH = $LIBS_PATH :\
$LD_LIBRARY_PATH ; program - name "
```

Listing 2: MPI singularity container bootstrap template

```
BootStrap: docker
From: ubuntu:xenial
#BootStrap: debootstrap
#OSVersion: xenial
#MirrorURL: http://us.archive.ubuntu.com/ubuntu/

%setup
######################
# ACTIONS FROM HOST
# use $SINGULARITY_ROOTFS to refer to container root (/)
######################

%post
######################
# INSTALL SECTION
######################

    #------------------
    # REQUERIMENTS
    #------------------

    REQUERIMENTS="openmpi-bin \
                  openmpi-common \
                  libopenmpi-dev \
                  dapl2-utils \
                  libdapl-dev \
                  libdapl2 \
                  libibverbs1 \
                  librdmacm1 \
                  libcxgb3-1 \
                  libipathverbs1 \
                  libmlx4-1 \
                  libmlx5-1 \
                  libmthca1 \
                  libnes1 \
                  libpmi0 \
                  libpmi0-dev"


    echo "Installing $REQUERIMENTS ..."
    apt-get update
    apt -y --allow-unauthenticated install $REQUERIMENTS

    mkdir -p /mnt
    mkdir -p /scratch

    #------------------
    # USER INSTALL
    #------------------

    # ... Install here your software


    #------------------
    # CLEAN APT files
    #------------------
    apt-get clean
    rm -rf /var/lib/apt/lists/*
    rm -rf /var/tmp/*

%runscript
######################
# RUN COMMAND SCRIPT
# Singularity "run" command launch this script
######################

    echo "Arguments received: $*"
    exec "$@"
```

performance of a docker container and a KVM virtual machine by running benchmarks such as Linpack, Stream, Fio, and Netperf. Ákos Kovács [25] compared different container techniques (Docker [7], Singularity [13], and LXC [12]) with KVM (Kernel-based Virtual Machine) virtualization and native OS performance using Sysbench for CPU benchmarking and IPerf tool for network benchmarking. Containers were almost at the level of the native performance for CPU benchmarks and close to native for network benchmarks. Á. Kovács et al. [28] used Docker and singularity containers to benchmark the throughput of a multi-path network library. Most common Docker container and a HPC specific Singularity container interconnected with twelve 100 Mbit/s links were used to evaluate the aggregation capabilities of the combination of these technologies. Saha et al. [33] presented a performance evaluation of Docker and Singularity on bare metal nodes in the Chameleon cloud in addition to a mechanism by which Docker containers can be mapped with InfiniBand hardware with RDMA communication. The performance analysis showed that scientific workloads for both Docker and Singularity based containers can achieve near-native performance. Diamanti published a container adoption benchmark survey [34] listing performance as one of the key challenges for running containers in production.

For the existing HPC container platforms: Singularity [10], [13] has build-in support for MPI (OpenMPI, MPICH, IntelMPI). Shifter [2] supports MPI depending on MPICH Application Binary Interface (ABI). Charliecloud [8] uses Slurm srun to run containers with MPI installed. To our knowledge, there is no study that proposed different approaches/alternatives for configuring/installing containers to work with MPI applications using one container platform.

IX. Conclusion

Traditional software integration and deployment performed in HPC systems is a time-consuming activity which relies on manual installations performed by system administrators and presents some issues which make it very inflexible. The main issue of the selected containerization technology, Singularity, relies on its integration with Open MPI. Different general approaches for enabling MPI containers on HPC clusters were proposed. We analyzed the use of CPU, RAM and network. The obtained parameters were very close to those obtained by the native execution of the same ones, reason why the loss is despicable. In addition, if we consider the time that will be saved using this novel form of software deployment, this minimal loss is more than justified.

X. Acknowledgments

This work has been supported and funded by the following projects:

- PRACE (Partnership for Advanced Computing in Europe), which is funded in part by the EU's Horizon 2020 Research and Innovation programme (2014-2020) under grant agreement 730913.
- MSO4SC (Mathematical Modelling, Simulation and Optimization for Societal Challenges with Scientific Computing) that has received funding from the EU's Horizon 2020 research and innovation programme under grant agreement 731063.
- HPC-Europe 3 project (Grant agreement 730897).

References

[1] B. Grüning, J. Chilton, J. Köster, R. Dale, N. Soranzo, M. van den Beek, J. Goecks, R. Backofen, A. Nekrutenko, and J. Taylor, "Practical computational reproducibility in the life sciences," *Cell Systems*, vol. 6, no. 6, pp. 631 – 635, 2018.

[2] D. Jacobsen and S. Canon, "Contain this, unleashing docker for hpc," in *Cray User Group 2015*, April 23, 2015.

[3] "Partnership for advanced computing in europe," http://www.prace-project.eu, accessed: 2019-10-16.

[4] K. Hoste, J. Timmerman, A. Georges, and S. D. Weirdt, "EasyBuild: Building software with ease," in *2012 SC Companion: High Performance Computing, Networking Storage and Analysis*. IEEE, Nov. 2012. [Online]. Available: https://doi.org/10.1109/sc.companion.2012.81

[5] "Marconi user documentation," https://wiki.u-gov.it/confluence/display/SCAIUS/UG3.1%3A+MARCONI+UserGuide, accessed: 2019-10-16.

[6] "Finisterrae supercomputer," https://www.cesga.es/en/infraestructuras/computacion/FinisTerrae2, accessed: 2019-10-16.

[7] D. Merkel, "Docker: Lightweight linux containers for consistent development and deployment," *Linux J.*, vol. 2014, no. 239, Mar. 2014. [Online]. Available: http://dl.acm.org/citation.cfm?id=2600239.2600241

[8] R. Priedhorsky and T. Randles, "Charliecloud: unprivileged containers for user-defined software stacks in hpc," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis on - SC 17*. ACM Press, 2017. [Online]. Available: https://doi.org/10.1145/3126908.3126925

[9] "Library and tool for running oci-based containers in pods," https://github.com/containers/libpod, accessed: 2019-10-16.

[10] G. M. Kurtzer, V. Sochat, and M. W. Bauer, "Singularity: Scientific containers for mobility of compute," *PLOS ONE*, vol. 12, no. 5, p. e0177459, May 2017. [Online]. Available: https://doi.org/10.1371/journal.pone.0177459

[11] J. Gomes, E. Bagnaschi, I. Campos, M. David, L. Alves, J. Martins, J. Pina, A. López-García, and P. Orviz, "Enabling rootless linux containers in multi-user environments: The udocker tool," *Computer Physics Communications*, vol. 232, pp. 84 – 97, 2018.

[12] "Lxc - linux containers," linuxcontainers.org/lxc, accessed: 2016-05-21.

[13] G. M. Kurtzer, "Singularity 2.1.2 - Linux application and environment containers for science," Aug. 2016. [Online]. Available: https://doi.org/10.5281/zenodo.60736

[14] "Quantum espresso," https://www.quantum-espresso.org/, accessed: 2019-10-16.

[15] "Openmpi," https://www.open-mpi.org/, accessed: 2019-04-01.

[16] "Intel mpi," https://software.intel.com/en-us/mpi-library, accessed: 2019-10-01.

[17] "Singularity admin guide for mpi," https://www.sylabs.io/guides/2.6/Admin-guide.pdf%20, accessed: 2018-12-16.

[18] "How does pmix work with containers?" https://pmix.org/support/faq/how-does-pmix-work-with-containers/, accessed: 2019-10-01.

[19] "Building containers for intel omni-path fabrics using docker* and singularity*," https://www.intel.com/content/dam/support/us/en/documents/network-and-i-o/fabric-products/Build_Containers_for_Intel_OPA_AN_J57474_v4_0.pdf, accessed: 2019-10-16.

[20] Abdulrahman Azab and Víctor Sande Veiga, "Mathematical, modeling and optimization for socital challenges with scientific computing - singularity use case," URL: https://docs.google.com/presentation/d/1Hi3tp0cVMwYaAy6rXgDlHejuuiX7s6hcyj2kW19vA7o, 12 2017.

[21] "Open mpi api/abi changes review - abi tracker project," https://abi-laboratory.pro/tracker/ timeline/openmpi/, accessed: 2019-10-16.

[22] "Quantum espresso benchmark," https://github.com/electronic-structure/benchmarks, accessed: 2019-10-16.

[23] "Intel math kernel library," https://software.intel.com/en-us/mkl, accessed: 2019-10-16.

[24] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio, "An updated performance comparison of virtual machines and linux containers," in *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, Mar. 2015. [Online]. Available: https://doi.org/10.1109/ispass.2015.7095802

[25] A. Kovacs, "Comparison of different linux containers," in *2017 40th International Conference on Telecommunications and Signal Processing (TSP)*. IEEE, Jul. 2017. [Online]. Available: https://doi.org/10.1109/tsp.2017.8075934

[26] M. Newlin, K. Smathers, and M. E. DeYoung, "ARC containers for AI workloads," in *Proceedings of the Practice and Experience in Advanced Research Computing on Rise of the Machines (learning) - PEARC 19*. ACM Press, 2019. [Online]. Available: https://doi.org/10.1145/3332186.3333048

[27] Y. Wang, R. T. Evans, and L. Huang, "Performant container support for HPC applications," in *Proceedings of the Practice and Experience in Advanced Research Computing on Rise of the Machines (learning) - PEARC 19*. ACM Press, 2019. [Online]. Available: https://doi.org/10.1145/3332186.3332226

[28] A. Kovacs and G. Lencse, "Evaluation of layer 3 multipath solutions using container technologies," in *2019 42nd International Conference on Telecommunications and Signal Processing (TSP)*. IEEE, Jul. 2019. [Online]. Available: https://doi.org/10.1109/tsp.2019.8768820

[29] C. Pahl, "Containerization and the PaaS cloud," *IEEE Cloud Computing*, vol. 2, no. 3, pp. 24–31, May 2015. [Online]. Available: https://doi.org/10.1109/mcc.2015.51

[30] S. Soltesz, H. Pötzl, M. E. Fiuczynski, A. Bavier, and L. Peterson, "Container-based operating system virtualization," *ACM SIGOPS Operating Systems Review*, vol. 41, no. 3, p. 275, Jun. 2007. [Online]. Available: https://doi.org/10.1145/1272998.1273025

[31] M. G. Xavier, M. V. Neves, and C. A. F. D. Rose, "A performance comparison of container-based virtualization systems for MapReduce clusters," in *2014 22nd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*. IEEE, Feb. 2014. [Online]. Available: https://doi.org/10.1109/pdp.2014.78

[32] J. Bhimani, Z. Yang, M. Leeser, and N. Mi, "Accelerating big data applications using lightweight virtualization framework on enterprise cloud," in *2017 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, Sep. 2017. [Online]. Available: https://doi.org/10.1109/hpec.2017.8091086

[33] P. Saha, A. Beltre, P. Uminski, and M. Govindaraju, "Evaluation of docker containers for scientific workloads in the cloud," in *Proceedings of the Practice and Experience on Advanced Research Computing - PEARC 18*. ACM Press, 2018. [Online]. Available: https://doi.org/10.1145/3219104.3229280

[34] "2019 container adoption benchmark survey," https://diamanti.com/wp-content/uploads/2019/06/Diamanti_2019_Container_Survey.pdf, accessed: 2019-10-16.