Available online at www.prace-ri.eu

Partnership for Advanced Computing in Europe

Evaluation of Linux Container and full virtualization for HPC Applications in PRACE 5IP

A. Azab [a*], G. Muscianisi [b], G. Wiber [c], C. Fernandez [d]

[a]University of Oslo, Oslo, Norway
[b] CINECA - Interuniversity Consortium, Italy
[c]French Alternative Energies and Atomic Energy Commission (CEA), France
[d]Fundación Pública Galega Centro Tecnolóxico de Supercomputación de Galicia (CESGA), Spain

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 that 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. Here we focus in PRACE-relevant HPC applications, including MPI and GPU applications, evaluated together with other collaborators from Europe and the USA. In addition to security and performance, PRACE virtualisation-service activity is working on solutions for the portability with templates and guidelines for building portable containers. Interesting and complementary to containers are fully-virtualised workloads running as VM jobs. Such solution is useful in cases where specific OS kernel/platform is required. The management of fully-virtualised workloads are also being considered and evaluated. Regarding security and performance, different container platforms (Docker, Singularity, and uDocker) have been evaluated in this white paper carried out under PRACE-5IP virtualisation service.

1. INTRODUCTION

Sharing of software packages is an essential demand among scientists and researchers in order to reproduce results [28]. 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 [1]. 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 [5].
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,

[*] *Corresponding Author: azab@mail.uio.no*
[1] *Those dependencies that are dependent on the OS or the OS release*

managing, and allocating containers. Figure 1 depicts structural comparison between native deployment, VMs, and Linux containers.

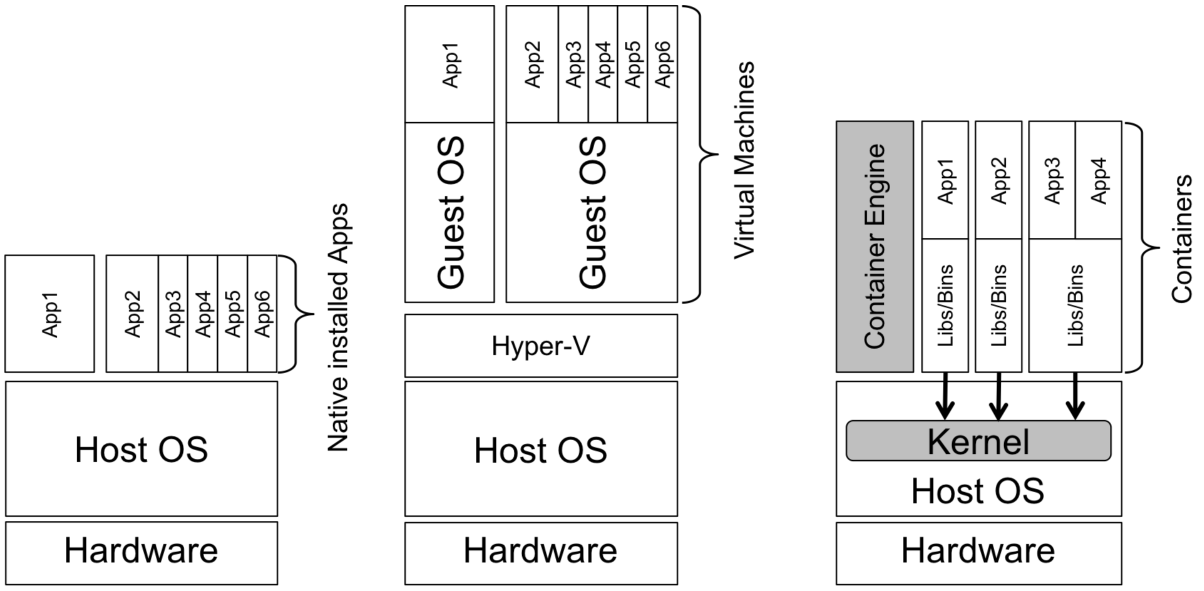

Figure 1. Native vs Virtual Machines vs Containers

Docker [4] is the most popular platform among users and IT centres for Linux containerization. A software tool can be packaged as a Docker image and pushed for sharing to public repository (Docker Hub). A Docker image can run as a container on any system that has a Linux kernel. Singularity [6] is another container engine, targeting mainly Linux containers on HPC platforms without root-privilege requirements and used on many HPC production clusters. With the use of Linux containers, researchers can install tools using their platform of choice, e.g. Ubuntu for bioinformaticians or CentOS for physicists as a Docker image and publish it on the Docker hub or just share the Docker file with collaborators (which can be used to build the image on any Docker engine). Then anyone who has a Docker engine and a Linux kernel may run the image and easily reproduce the same functionality. This arrangement makes life easier for software developers in that they no longer need to write multiple installation guides and test on different Linux distributions. It also makes life easier for system administrators, as instead of receiving software requests of type: "I need software X, and here is the installation guide, please install it!” requests will be of type: "I need software X, here is the name of its Docker image, please pull and test it". In addition, no software maintenance will be needed and no dependency conflicts will arise when installing new software [2]. Container platforms that target HPC systems, e.g. Singularity and Shifter, made it possible to use Docker containers in production for HPC systems.
Virtual machines (VMs) are widely adopted as a software packaging method for sharing collections of tools, e.g. BioLinux [3]. Each VM contains its own operating system, known as the guest OS, on the top of which software packages are installed. A VM hypervisor, is the platform for managing and monitoring VMs. VM technology is suitable for packaging large collections of tools that run on the top of a specific OS platform, e.g. a GUI that runs python and R tools on the top of Ubuntu Linux. VMs are also effective in cases where an application needs a specific Linux kernel or Windows kernel.

2. DOCKER

Docker is a tool that automates the deployment of applications inside software containers. It provides an additional layer of abstraction (through the Docker engine) and operating-system-level virtualization on Linux [4]. Docker is not a new technology itself, but is a high-level tool which was built based on LXC Linux containers API. Then Docker developed its own container runtime “containerd” [29], which has become independent from Docker. Other examples of Linux containers are LXD, CGManager, and LXCFS [17]. The main feature of Docker containers, and Linux containers in general, is that they offer portability similar to VMs but without the overhead of running a separate kernel and simulating the hardware [17]. Docker containers are executed and controlled by the Docker engine (Docker daemon). Since it does not include a full guest OS, a Docker container is smaller and faster to start up than a VM. A Docker container instead mounts a separate root file-system, which contains the directory structure of the Unix-like OS plus all the configuration files, binaries and libraries required to run user applications. The boot file-system which contains the bootloader and the kernel is not included in the container, but a container uses the host's boot file-system [18]. When Docker mounts the container's root file-system, it starts read-only, as in a traditional Linux boot, but then, instead of changing the file system to read-write mode, it uses

union mount [19] to add a read-write file system layer above the read-only file system. It is also possible to have multiple read-only file system layers stacked on top of each other, see Figure 2. Each of those file-systems can be considered as a layer [20].
Based on the above description, one could start with e.g. a Debian base image (which contains only the Debian root file-system), and run it from the Docker engine on a machine with e.g. Red Hat Enterprise Linux (RHEL) kernel. Then install a piece of software, e.g. emacs, on the top of this base image, and deploy this as a new image (containing both Debian root file-system and emacs installed). Using this new image as a base image, one could install another software package, e.g. Apache server, and make another image which has both emacs and Apache installed on Debian, and so on.

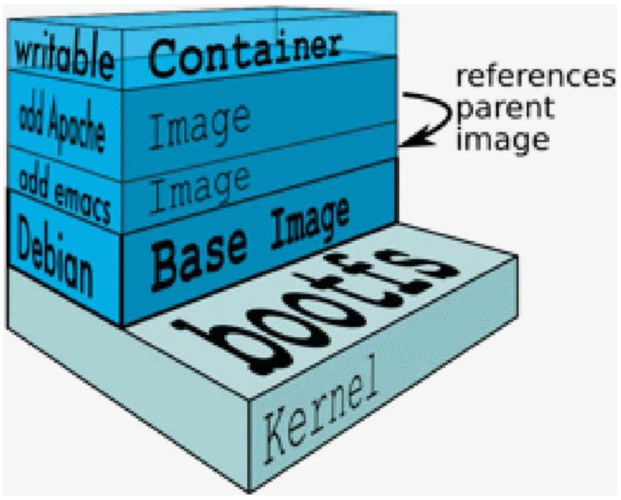

Figure 2. Docker File-system layers example

2.1 BUILDING A DOCKER IMAGE

Building own Docker image is performed in one of two scenarios, presented in Figure 3:

- Interactive building which is carried out by starting a base image as a container (via 'Docker run'), running the commands to install the desired software on the running container, then committing the changes creating a new image on the local Docker repository.
- Dockerfile based building, where a building script "Dockerfile" is written using a specific format, and the image is built using the docker build command and the Dockerfile as an argument

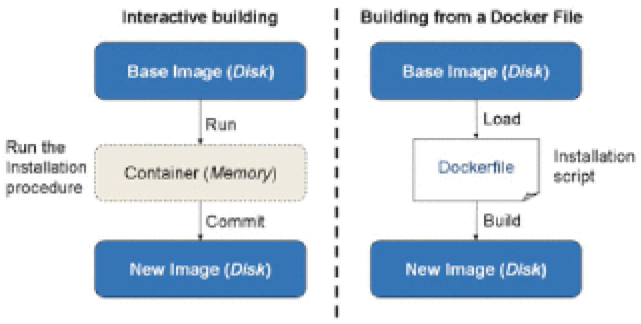

Figure 3. *Building a Docker image: Interactive based vs Dockerfile based*

3. RELATED WORK

Numerous efforts have been carried out in the direction of enabling Docker containers on HPC platforms. Docker Swarm [9] is the container orchestration platform offered by Docker. Swarm offers great scalability, but it is very poor in terms of security. In addition, the scheduling and resource management is too basic which makes it insufficient for large HPC systems in production. Google's Kubernetes [10] is a competitor to Docker Swarm as another docker container orchestrator. It offers more control over running containers, in addition to self-healing, but it is still poor in terms of scheduling and resource management. In addition, it is very complex to install and maintain.

A common HPC integration problem for both Docker Swarm and Kubernetes is that they are both designed to be standalone queuing systems, not runners or wrappers. So, one cannot have Swarm or Kubernetes plugged into already installed queuing system, e.g. TORQUE or Slurm, but has to replace the entire queuing system. HTCondor [11] has recently provided support for Docker through the *docker universe* [12]. HTCondor is more trustworthy for HPC systems since it is a well-known and well tested platform for both HPC and HTC. We have deployed Docker with HTCondor on a test environment, and the implementation was proven to be very user friendly. But it lacks flexibility, e.g. in terms of mounting volumes. In addition, it does not have any control over resource usage by the running containers.
Shifter [13] is a platform to run Docker containers on Slurm. It is developed by the National Energy Research Scientific Computing Center (NERSC) and is deployed in production on a Slurm cluster of Cray supercomputers. Shifter is however not following the Docker standards and is not using the Docker engine for running and managing containers. It has 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 formatted images. Previously NERSC introduced MyDock [13] 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 initially running as root and mounting `/etc/passwd` and `/etc/group` in each single container, then lets the user access the container as him/herself. Socker adopts a more secure and less complex strategy by running containers as the user from the very beginning, avoiding any threats that may result in running containers as root. Similar to singularity, shifter being immature, and relying on its own image format and own engine, doesn't make it sufficiently flexible for large production systems. Developing a runner that uses the Docker engine is more realistic since Docker is a well known and maintained platform in addition to being used by millions of users and a number of IT research support centers worldwide.

4. PROTOTYPES

PRACE-5IP virtualization service has targeted evaluation and benchmarking of containerized and fully virtualized workloads on both bare-metal and cloud-based HPC clusters. The service includes the following prototypes:

- Container workloads. Prototypes:
    - Docker: Socker and uDocker
    - Singularity (with MPI and GPU workloads)
- Fully virtualized workloads on Slurm. Prototype:
    - PCOCC
- Containerized cluster with web-based front-end. Prototype:
    - Containerized Galaxy-HTCondor
- Container enabled meta-Scheduler. Prototype:
    - ARC Control Tower (aCT)

4.1 SOCKER: SECURE DOCKER

Before deciding to deploy Docker containers on a queuing system in production, there are two main issues to resolve: First, the default configuration of Docker is to run containers as root. For the container process to be bounded with the user privileges and for sys-admins to keep record of who is running what, the container process must run as the submitting user. Second, the resource consumption (CPU and memory) of a Docker container running inside a job must be bounded with the limitations for resource consumption set by the queuing system for this particular job.

Socker is a secure wrapper for running Docker containers on Slurm [7] and similar queuing systems, e.g. Moab [8] and PBS [14]. Socker is tested by the research computing group at the University Center for Information Technology, USIT at UiO. Socker is not a replacement of the Docker engine for running and managing containers. It runs Docker containers as the submitting user in addition to enforcing the membership of a running container, as part of a HPC job, in the cgroups [15] assigned by the queuing system to this job. Basic testing of Socker has been carried out to run MPI jobs on a Slurm cluster. Socker has also been tested for Many-Task computing (MTC) [16] running container jobs on a system of interconnected clusters. Socker has proven to be secure, in addition to introducing almost no computational overhead. Socker has been also tested for High-Throughput computing running submitting container jobs to interconnected clusters. Further developments of Socker are ongoing, and it is now available on github [2].

Figure 4 shows the structure of Socker. Users are categorized into system administrators and regular users. System administrators are given privileges to run the docker command, i.e. members of the docker group. Regular users can run Docker only through Socker.

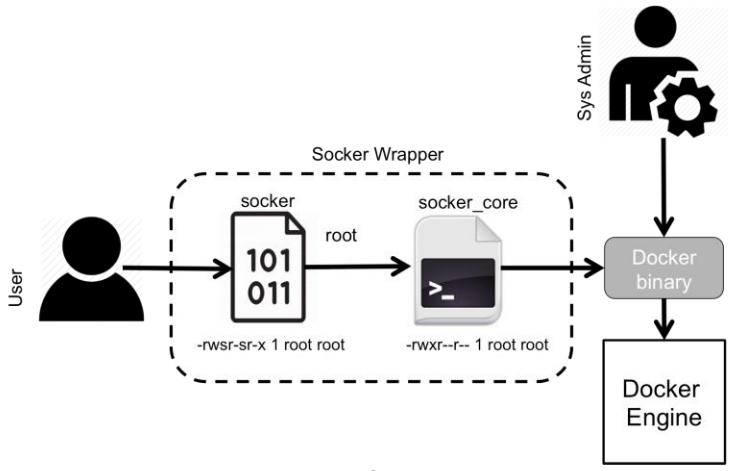

Figure 4. Socker wrapper

To enable Socker on HPC clusters, Docker 1.7 has been installed on compute nodes on the main cluster at UiO, Abel [21]. Currently Docker doesn't support shared image repositories among hosts, so each compute node has it's local image repository (layered filesystem).

MPI support

Socker doesn't have implicit MPI support so far. It can be used with MPI by doing the following:

- Install MPI (and IB drivers if any) on the host
- Use a Docker container with MPI supporting application (and the IB driver installed if any)
- Run uDocker with MPI as:

```
mpiexec -np <N> socker run -e LD_LIBRARY_PATH=/usr/lib <app> <args>
```

Use cases

The following use cases have been operated with Socker on Abel [21] and Colossus [22] clusters:

- *Data analytics:* **mriqc**, **freesurfer**, **heudiconv**
- *Physics:* **GAMBIT**
- *Robotics:* **Gazebo**

The following genomic tools has been supported for the Tryggve [23] project: **HTSeq [26]**, **BWA-MEM [27]**, **GATK [28]**, and **subread [29]**

To evaluate the performance of Socker for HPC applications, we performed an experiment on our main Slurm cluster, Abel. The aim of this experiment was to indicate whether Socker introduces considerable computational overhead. We used one image from Docker hub, genomicpariscentre/bowtie1, for running sequence alignment using Bowtie [24]. The input dataset used is ≃5 GiB Fastq file, to be aligned against human genome hg19. We used the pMap package [25] to divide the alignment into parallel MPI jobs. Three compute nodes (2*16 + 20 cores) were reserved for this docker test. The experiment included the following:

- Alignment with pMap using Native Bowtie binary.
- pMap running the Bowtie Docker image using docker run as a system administrator account that is a docker group member.
- pMap running the Bowtie Docker image using Socker and a regular user account.

The Bowtie image has been pulled on the three nodes in advance, to avoid any additional latency pulling the image from the Docker hub. All Docker runs (both direct Docker and Socker) were configured to remove the container after exiting. Figure 5 presents the total run time against the number of parallel processes. 1, 8, 16, and 32 processes were used. It is clear that running container jobs (for both Docker and Socker) introduces very small overhead compared to running native jobs. It can be observed that the overhead grows with increasing the number of parallel processes. This results from: 1) the overhead of starting new containers, 2) Removing exited containers is enabled, which introduces additional overhead (but is necessary for cleaning up and saving disk space on compute nodes). It is also noticed that Socker overall introduces almost no additional overhead compared to docker run.

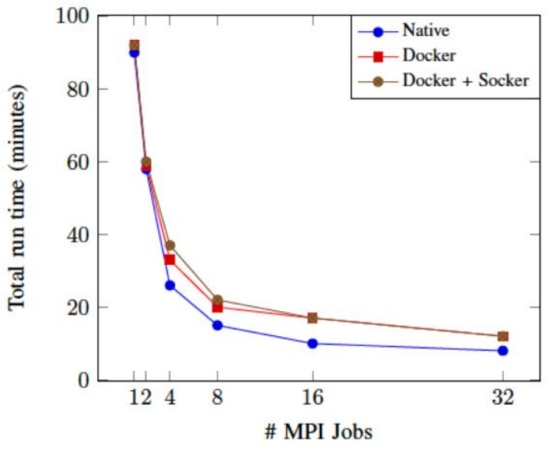

Figure 5. Total run time of parallel pmap MPI jobs using bowtie aligner as: Bowtie native binary, bowtie docker image using docker run, and bowtie Docker image using Socker

4.2 PCOCC: PRIVATE CLOUD ON A COMPUTE CLUSTER

PCOCC [31] (Private Cloud on a Compute Cluster) allows users of HPC clusters to host their own clusters of VMs on compute nodes alongside regular jobs. Users are thus able to fully customize their software environments for development, testing or facilitating application deployment. Compute nodes remain managed by the batch scheduler as usual, since the clusters of VMs are seen as regular jobs. From the point of view of the batch scheduler, each VM is a task for which it allocates the requested CPUs and memory and the resource usage is billed to the user, just as for any other job. For each virtual cluster, PCOCC instantiates private networks isolated from the host networks, creates temporary disk images from the selected templates (using Copy-on-Write) and instantiates the requested VMs. PCOCC is able to run virtual clusters of several thousands of VMs and has enabled varied new uses of CEA's compute clusters, from running complex software stacks packaged in an image to reproducing Lustre issues happening at large scale without impacting production servers.

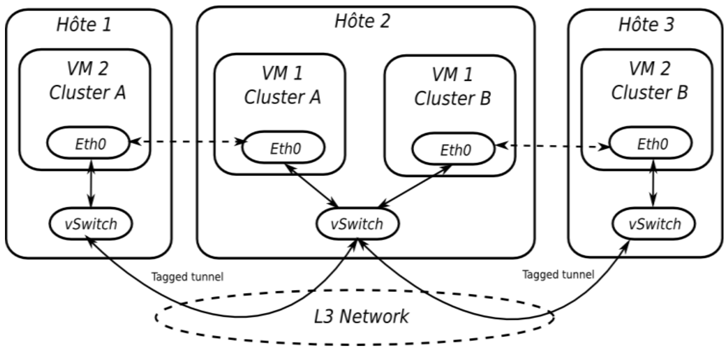

Figure 6. PCOCC Networking model

Benchmarking Results

To promote such a tool, performances are critical. So several tests using VMs instead of bare-metal hardware have been done.

- **First Infiniband tests: IMB tests.** Figure 7 shows negligible impact on the bandwidth. Some overhead has been seen on the latency: less than 500ns
- **Parallel benchmarks:** As shown in Figure 8, in general, these benchmarks show less than 5% overhead using VM compare to bare-metal hardware. In some cases, VMs are faster than traditional compute nodes: thanks to huge pages. From an I/O point of view, benchmarks are very sensitive to the size of the I/O and the result depend really of how I/O are done.

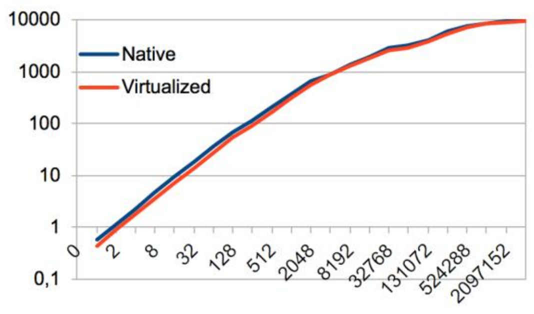

Figure 7. PCOCC Infiniband benchmark

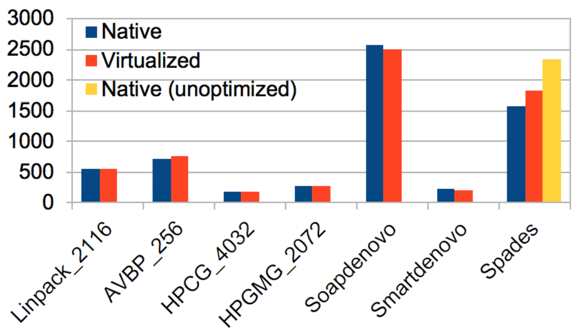

Figure 8. PCOCC Parallel benchmarks

- **Other tests:** PCOCC is also very efficient to launch VMs: more than 1024 VMs CentOS7 in less than 1 minute has been reached (1 VM / core). Moreover, a checkpoint of 4032 cores (1.7GB/core) of the Linpack benchmark, has been written in less than 5 min for 6.8TB of checkpoint data.

4.3 SINGULARITY

Released officially for the first time in 2016, Singularity [6] is a container paradigm, designed for HPC platforms, that aims to reach three specific goals: reproducibility of results, mobility of compute and user freedom.
To reach these goals, Singularity developers were able to achieve containers as a single file image that can be easily copied or transferred between machines. In such an image file: the operation system, the packages, the libraries, and needed software are all packaged together in a single file. Moreover, these Singularity containers can interact with files local to the container in the directory where the container is executed (binding option).
Since Singularity containers are designed for HPC platforms, it is possible in a very simple way to run pure MPI or hybrid (MPI + OpenMP) applications inside a singularity container. Also GPU support is available in these containers, by installing the required driver and toolkit.
Applications which run in a singularity container, run with the same "distance" to the host kernel and hardware as natively running applications. Singularity launches the container as the calling user in the appropriate process context. There is no root daemon process and no escalation of privileges within the container, as in the case of Docker containers.
Even if Singularity is starting to be widely used in a very short time, the most popular container system today is arguably Docker. This is not an issue, it's not necessary to convert a previously created Docker container in Singularity one, because Singularity is designed in a way that it is easy to use a previously created Docker container only by bootstraping such a Docker container inside singularity.
The issue with singularity that it is still not as popular as Docker among the scientific community which we are serving. It is not even any close to. Singularity supports conversion of Docker images to singularity images. But after deeply testing this feature, it is still not stable. Many conversion attempts failed miserably. Therefore we cannot rely on singularity for production deployment until it proves that all docker containers, or at least most, can be converted to singularity containers.

MPI support

Singularity containers can be created for any application and are not just limited to HPC applications. However, one of the important features for HPC is that Singularity can do is create simple containers for MPI and MPI+OpenMP applications. These have been very difficult for other container systems to handle, but Singularity treats the applications like an executable, so the "mpirun" command treats it like any other executable.
Singularity is still developing, but it can already be used to create and run containers for large MPI applications. The main issues are related to the unsupported backward compatibility in OpenMPI 2.x So actually, until 3.x will, the same version of OpenMPI in the singularity container and in the host have to be installed. For OpenMPI 3.x+, host MPI doesn't need to be the same as the container MPI. Moreover, for large MPI applications running on more than one node, the needed driver for interconnection network, as Infiniband, has to be installed inside the container.

The Singularity team has created registries; SingularityHub, a public registry like DockerHub, and Sregistry. Both are tools used to remotely store and transfer Singularity images from the Cloud. While SingularityHub is hosted and maintained by the Singularity team, Sregistry can be deployed and managed in our own cloud. In addition, the Singularity tool has included a new pull command for downloading or using remote images stored by means of these services. We can take advantage of these improvements to enrich application workflows in two ways;

delivery automation and workflow portability. The need to be superuser to create Singularity containers is still present, and it will remain as an inherent requirement in the implemented containerization model. In multi user systems as HPCs, a normal user will usually not have superuser permissions. Taking all this into account, the Singularity usage workflow has been updated to be adapted to its new features.

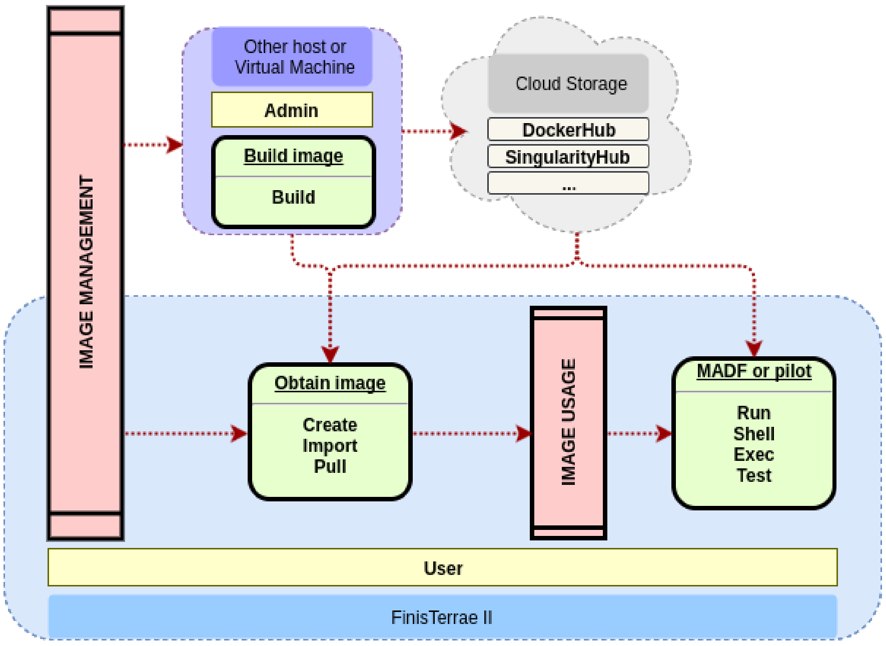

Figure 9. Workflow of Singularity images in the e-Infrastructure

Figure 9. 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 run 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. At EPCC the tests included the ARCHER benchmark suite (https://github.com/hpc-uk/archer-benchmarks).

Use cases and benchmarking results at FinisTerrae II

Here we describe the use cases at FT2. 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. 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 the de facto industry standard for measuring sustained RAM 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.

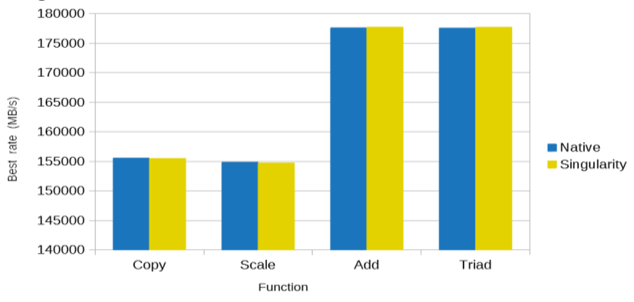

Figure 10. STREAM best bandwidth rates comparison

As we can see in Figure 10, 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 a 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.

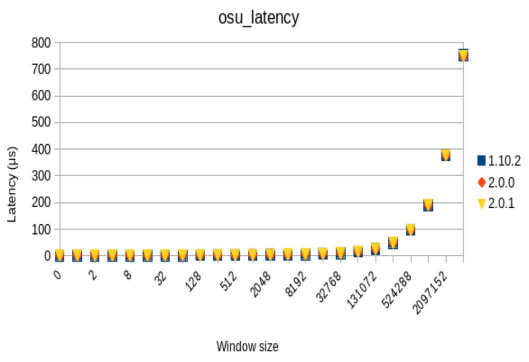

Figure 11. Latency from Singularity using OSU micro-benchmark

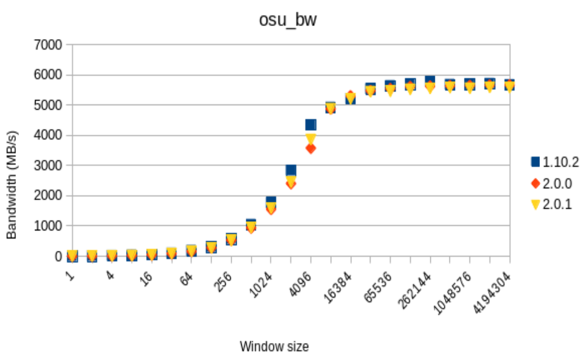

Figure 12. Bandwidth from Singularity using OSU micro-benchmarks

We can see in figure 11, 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 figure 12, we can observe that the general is as expected. The maximum bandwidth reached is close to 6GB/s, which are again in a correct value ranges for Infiniband (FDR generation). 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 full advantage of these HPC resources under the specified conditions.

Best practices building portable containers

The following are best practices for building Singularity containers. These instructions provide the keys to build valid containers and avoid issues, making the containers completely transparent for users and also for the e-Infrastructure:

1. Provide a well-tested Singularity container. Within the container, an entire distribution of Linux or a very lightweight tuned set of packages can be included, preserving the usual Linux directories hierarchy. It's also recommended to set up the required environment variables within the container in order to expose a consistent environment.
2. To get transparent access to the host e-infrastructure storage from the containers, the shared filesystem root directories, e.g. /shared must exist within the container to be shared with the host.
3. To run parallel applications using multiple nodes with MPI, the container must have installed support for MPI and Process Management Interfacr (PMI). Due to the Singularity hybrid MPI approach, it's mandatory to use the same implementation and version of MPI installed at the host and inside the container to run MPI parallel applications. Also for taking advantage of some HPC resources like Infiniband networks or GPUs, the container must support them. This means that the container must have installed the proper libraries to communicate with the hardware and also to perform inter-process communications.
4. In the case of Infiniband support, there are not any known restrictions about the Infiniband libraries installed inside the container. In the particular case of using GPUs from a Singuarity container, the contained Nvidia driver must exactly match the driver installed at the host. Singularity provides GPU containers portability through the experimental Nvidia support option to allow containers to automatically use the host drivers.

UDOCKER

uDocker is a basic user tool to execute simple docker containers in user space without requiring root privileges. It enables download and execution of docker containers by non-privileged users in Linux systems where docker is not available. It can be used to pull and execute 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 "executes" the containers by simply providing a chroot like environment over the extracted container. The current implementation supports different methods to mimic chroot enabling execution of containers without requiring privileges under a chroot like environment. Udocker transparently supports several methods to execute the containers using tools and libraries such as:

Proot, Fakechroot, runC, Singularity

uDocker has been successfully tested with MPI applications at FinisTerrae II cluster at CESGA. The deployment included the following procedure:

- Install compatible OpenMPI version
- Install hardware related libraries: network drivers (infiniband, mellanox, etc.)
- Compile the MPI test
- Expose some host things into the container: mpirun -np P udocker run –hostenv –hostauth MYCONTAINER MYAPP

The following is a programmatic template to make uDocker (devel) work with NVIDIA GPUs also in FinisTerrae II:

```
curl https://raw.githubusercontent.com/mariojmdavid/udocker/devel/udocker.p
y > bin/udocker
chmod 755 bin/udocker

udocker pull mariojmdavid/tensorflow-1.5.0-gpu
udocker create –name=tf mariojmdavid/tensorflow-1.5.0-gpu

PATH=${HOME}/bin:$PATH
UDOCKER_DIR=${HOME}/.udocker
CONT=tf
WDIR=`pwd`

echo $PATH
echo $UDOCKER_DIR
echo "----------------"

echo "Doing the setup"
udocker setup –execmode=F3 –nvidia ${CONT}

echo "RUNNING"
udocker run -w /home/tf-benchmarks ${CONT} \
python /home/tf-benchmarks/benchmark_alexnet.py >> tf.out
```

5. SUPPORTED USE CASES

PRACE virtualization service has published a web-form [30]for collecting research use cases for containers and VMs in HPC. The following is a brief description of the 15 supported use cases, which are categorised as follows:

- **Name of the software:** caffe, ROS, gazebo, FeniCS, mriqc, freesurfer, heudiconv, upc,upc++, gasnet intel PCM (performance counter monitor), GAMBIT, FeniCS, ARMplusplus, anvi'o, GAMBIT, Ubuntu (OS).
- **Purpose of the software:** Data analytics: 7 (46.7%), Virtualization: 3 (20%), Deep learning: 4 (26,7%), Machine learning: 1 (6.7%), and Other: 5 (33%)
- **Type of packaging:** Virtual Machine: 5 (33.3%), Containers: 10 (66.7%).
- **Support for parallelization:** Yes: 13 (86.7%), No: 2 (13.3%)
- **Kind of parallelization:** Shared memory: 9 (60%), Distributed memory: 9 (60%)

- **The approximate number of researchers using the software:** Less than 5: 1 (6.7%), Between 5 and 20: 10 (66.7%), More than 20:4 (26.7%)

Commenting on the use case description:
- The majority of use cases are for containers.
- Most use cases are for software that support parallelization.
- About four use cases are useful for more than 20 researchers.

6. CONCLUSIONS

Several application prototypes with different container platforms have been tested and evaluated under different use cases. In general, containers have been proven efficient and useful for building portable applications. Portability is not complete for MPI and GPU applications, for this, best practices in section 4.3 should be followed. Singularity is the most suitable container platform for HPC clusters. The Socker prototype proved that the usage of the Docker engine to run docker containers can be secured. uDocker is another solution for unprivileged docker containers. The solution lacks maturity for MPI applications. PCOCC is a production-ready tool for managing VM workloads. PCOCC is used in production at CEA by different communities. Container support in PCOCC is currently supported for pilot users.

Acknowledgements

This work was financially supported by the PRACE project funded in part by the EU's Horizon 2020 Research and Innovation programme (2014-2020) under grant agreement 730913.

References

[1] http://www.prace-project.eu

[2] Azab, Abdulrahman. Enabling Docker Containers for High-Performance and Many-Task Computing , In *2017 IEEE International Conference on Cloud Engineering (IC2E).* IEEE Computer Society. S 279 – 285

[3] bio-linux overview. http://environmentalomics.org/bio-linux/

[4] D. Merkel, Docker: Lightweight linux containers for consistent development and deployment,» Linux J., vol. 2014, no. 239, Mar. 2014.

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

[6] 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

[7] M. A. Jette, A. B. Yoo, and 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.

[8] "Layers," https://docs.docker.com/terms/layer/, accessed: 2015-04-22. "Moab hpc suite," http://www.adaptivecomputing.com/products/hpcproducts/moab-hpc-basic-edition/, accessed: 2017-01-21.

[9] "Docker swarm," https://docs.docker.com/swarm/, accessed: 2016-05-21.

[10] D. K. Rensin, Kubernetes – Scheduling the Future at Cloud Scale, 1005 Gravenstein Highway North Sebastopol, CA 95472, 2015. [Online]. Available: http://www.oreilly.com/webops-perf/free/kubernetes.csp

[11] M. Litzkow, M. Livny, and M. Mutka, "Condor – a hunter of idle workstations," in Proceedings of the 8th International Conference of Distributed Computing Systems, June 1988.

[12] "Docker and htcondor," https://research.cs.wisc.edu/htcondor/ HTCondorWeek2015/presentations/ThainG Docker.pdf, accessed: 2016-05-21.

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

[14] H. Feng, V. Misra, and D. Rubenstein, "Pbs: A unified priority-based scheduler," ser. SIGMETRICS '07. New York, NY, USA: ACM, 2007,pp. 203–214.

[15] J. Corbet, "Notes from a container," https://lwn.net/Articles/256389/, October 29, 2007, accessed: 2016-05-21.

[16] I. Raicu, I. T. Foster, and Y. Zhao, "Many-task computing for grids and supercomputers," in Many-Task Computing on Grids and Supercomputers, 2008. MTAGS 2008. Workshop on. IEEE, Nov. 2008, pp. 1–11.

[17] "Infrastructure for container projects", 04 2015, [online] Available: https:lllinuxcontainers.org/.

[18] “File System”, 04 2015, [online] Available: http://docs.docker.com/terms/filesysteml.
[19] Pendry Jan-Simon, Marshall Kirk McKusick, “Union Mounts in 4.4BSD-Lite”, Proceedings of the USENIX Technical Conference on UNIX and Advanced Computing Systems, vol. 2533, December, 1995.
[20] “Layers”, 04 2015, [online] Available: https:lldocs.docker.com/terms/layer/.
[21] “The Abel supercomputer”, [online] Available: https://www.uio.no/english/services/it/research/hpc/abel/
[22] Tjenester for sensitive data (tsd), [online] Available: http://www.uio.no/tjenester/it/forskning/sensitiv/.
[23] Neic tryggve: Collaboration on sensitive data, [online] Available: https://wiki.neic.no/wiki/Tryggve.
[24] B. Langmead, C. Trapnell, M. Pop, S. Salzberg, “Ultrafast and memory-efficient alignment of short dna sequences to the human genome”, Genome Biology, vol. 10, no. 3, pp. R25, 2009.
[25] pmap: Parallel sequence mapping tool, [online] Available: http://bmi.osu.edu/hpc/software/pmap/pmap.html.
[26] “HTSeq: Analysing high-throughput sequencing data with Python”, [online] Available: https://htseq.readthedocs.io/en/release_0.11.1/
[27] H Li “Aligning sequence reads, clone sequences and assembly contigs with BWA-MEM”, arXiv preprint arXiv:1303.3997
[28] Bjorn Gruning et al. “Practical computational reproducibility in the life sciences”, Cell Systems doi: 10.1016/j.cels.2018.03.014
[29] “An industry-standard container runtime with an emphasis on simplicity, robustness and portability”, [online] Available: https://containerd.io/
[30] https://skjema.uio.no/prace-containers
[31] “Run VMs on an HPC cluster”, [online] Available: https://github.com/cea-hpc/pcocc

APPENDIX

Socker installation

- Install Nuitka (https://nuitka.net/) with its prerequisites (python and gcc)
- Compile socker:

```
  nuitka --recurse-on socker.py
```

- Change the owner of the binary to root and enable SUID:

```
mv socker.exe socker
sudo chown 0:0 socker
sudo chmod +s socker
```

- Create a list of authorized images as root (you need to fix the path to the images file in socker before compiting):

```
  sudo vim socker-images
```

- Options:

```
socker --help

NAME
  socker - Secure runner for Docker containers

SYNOPSIS
  socker run <docker-image> <command>

OPTIONS
  --version
          show the version number and exit
  -h, --help
          show this help message and exit
  -v, --verbose
          run in verbose mode
  images
          List the authorized Docker images (found in socker-images)
  run IMAGE COMMAND
          start a container from IMAGE executing COMMAND as the user
```

```
EXAMPLES
  List available images
          $ socker images
  Run a CentOS container and print the system release
          $ socker run centos cat /etc/system-release
  Run the previous command in verbose mode
          $ socker -v run centos cat /etc/system-release

SUPPORT
  Contact hpc-drift@usit.uio.no
```