# D12.3 – Using container technologies to improve portability of applications in HPC

| | |
|---|---|
| Deliverable No.: | D12.3 |
| Deliverable Name: | Using containers technologies to improve portability of applications in HPC |
| Contractual Submission Date: | 30/04/2019 |
| Actual Submission Date: | 29/04/2019 |
| Version: | V3.0 |

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 730897.

| *COVER AND CONTROL PAGE OF DOCUMENT* | |
|---|---|
| Project Acronym: | **HPC-Europa3** |
| Project Full Name: | Transnational Access Programme for a Pan-European Network of HPC Research Infrastructures and Laboratories for scientific computing |
| Deliverable No.: | D12.3 |
| Document name: | Using containers technologies to improve portability of applications in HPC |
| Nature (R, P, D, O): | R |
| Dissemination Level (PU, PP, RE, CO): | PU |
| Version: | V3.0 |
| Actual Submission Date: | 30/04/2019 |
| Author, Institution:<br>E-Mail: | Giuseppa Muscianisi, CINECA:<br>g.muscianisi@cineca.it |
| Other contributors | Raül Sirvent, Marta Garcia, Oleksandr Rudyy (BSC)<br>Niall Wilson (NUIG-ICHEC)<br>Atte Sillanpää, Ari-Matti Saren (CSC)<br>Simone Marocchi (CINECA) |

**ABSTRACT:**
The aim of this deliverable is to provide indications about how to use container to improve portability of applications on HPC systems.
Important steps to run a containerized application in a HPC system include the combination of a container instance within the job batch scheduler available on the system. In the first sections, this deliverable is focused about the usage of MPI libraries, network interconnection drivers and accelerators within a container. Then, some considerations about the integration with workload managers are provided, together with an optimization for Singularity containers.
Finally, a case study about containerization in cloud environment is reported.

**KEYWORD LIST:**
High Performance Computing, HPC, Virtualization, Containers, Docker, Singularity, MPI, GPU, SLURM

| *MODIFICATION CONTROL* | | | |
|---|---|---|---|
| **Version** | **Date** | **Status** | **Author** |
| 1.0 | 05/04/2019 | Complete draft ready | Giuseppa Muscianisi |
| 2.0 | 23/04/2019 | After reviewers's comments | Giuseppa Muscianisi |
| 3.0 | 26/04/2019 | After CSC corrections | Giuseppa Muscianisi, Atte Sillanpää |

*TABLE OF CONTENTS*

Glossary ..... 4
Executive summary ..... 5
1 Introduction ..... 6
2 Application packaging in containers ..... 7
3 Containerization of Parallel MPI application ..... 8
3.1 Singularity and Open MPI ..... 8
3.2 Singularity and Intel MPI ..... 10
4 Networking libraries in a container ..... 11
4.1 Intel Omni-Path Fabrics in a container ..... 11
4.1.1 Quantum Espresso singularity container test case on Cineca MARCONI system ..... 12
5 GPUs utilization within a container ..... 16
5.1 Tensorflow singularity container test case on Cineca GALILEO system ..... 16
6 Integration with Batch Queuing Systems ..... 19
6.1 Integrating containers with workload managers ..... 19
6.2 Performance considerations with workload managers ..... 20
7 Optimizations identified ..... 22
7.1 Singularity integration with MPI ..... 22
8 Application containerization using cloud resources ..... 23
8.1 Utilizing containers in Elmer development and production – Case study ..... 24
9 Conclusion ..... 25
References ..... 26

# Glossary

| Abbreviation | Description |
|---|---|
| FFTW | Fastest Fourier Transform in the West |
| FPGA | Field-Programmable Gate Array |
| GPFS | General Parallel File System |
| GSS | GPFS Storage Server |
| GPU | General Processing Unit |
| HPC | High Performance Computing |
| Intel MIC | Intel Many Integrated Core |
| Intel OPA | Intel Omni-Path Architecture |
| IT | Information Technology |
| IPoIB | Internet Protocol over Infiniband |
| MPI | Message Passing Interface |
| OFED | OpenFabrics Enterprise Distribution |
| OPA | Omni-Path Architecture |
| OpenMP | Open Multiprocessing |
| ORTED | Open RTE daemon |
| OS | Operative System |
| PBS Pro | Portable Batch System Professional |
| PMI | Process Management Interface |
| PMIx | Process Management Interface for Exascale Environments |
| PWSCF | Plane-Wave Self-Consistent Field |
| RDMA | Remote direct memory access |
| SLURM | Simple Linux Utility for Resource Management |
| SSD | Solid State Drive |
| VM | Virtual Machine |

# Executive summary

The aim of this deliverable is to provide indications about how to use containers to improve portability of applications on HPC systems.

Important steps to run a containerized application in a HPC environment include the combination of a container instance within the job batch scheduler available on the system. In the first sections, this deliverable is focused about the usage of MPI libraries, network interconnection drivers and accelerators within a container. Then, some considerations about the integration with workload managers are provided, together with an optimization for Singularity containers.

Finally, a case study about containerization in cloud environment is reported.

# 1 Introduction

In the framework of HPC-Europa3 project, researchers have the opportunity to access different supercomputer infrastructures for their computations, both for using a different architecture from their home one and for working side by side to other researchers of the same computational field establishing an international collaboration. As the architecture of the supercomputing infrastructure may change from site to site, mechanisms are needed to help researchers to deploy their applications so to maximize the time of their visit. The installation and configuration of an application with all its dependencies on a supercomputer could require a significant amount effort and time, due to the operative system, libraries, tools needed to run the application.

In this context, container technologies can be used to simplify such installation process: researchers who have the possibility to access a supercomputer machine can build their own container, packaging all the needed applications, libraries, tools and configuration files within, and run this container directly on the supercomputer with minor or even no additional effort compared to a bare metal installation.

It is important to consider that supercomputing centres operate clusters with different compute node architectures, interconnectivities, storages, accelerators and so on. So, it is very difficult to provide the users with updated information and documentation about, 1) possible compatibility issues using different Message Passing Interface (MPI) libraries, 2) General Processing Units (GPUs) and other accelerator utilizations, 3) networking drivers installation, and 4) integration with traditional batch job scheduler.

The aim of the Task 12.2 in HPC-Europa3 project is to describe how it might be possible to improve portability of applications in HPC using container technologies. This goal is reached by analysing all the previous points and providing indications for the final user.

Relying on the evaluations reported in deliverable D12.1 [1], in the next sections considerations are reported about the deploying and running of a containerized application in HPC environments. The deliverable is mainly focused on Singularity [2] and Docker [3] technologies, Docker being the most used containerization tool and Singularity the widely used containerization tool suitable for HPC environment. The most important feature of Singularity is to be “secure” in the sense that the container execution takes place in the user space and it is able to run Docker container in the user space, without having the Docker daemon running on the host.

After a note about the usage of package managers Spack [4] and Easybuild [5] to build containers, a discussion about how run parallel applications will be done focusing in particular about Open MPI [7] and Intel MPI [8] integration in Singularity. Since a HPC system is usually characterized by having a high-bandwidth low-latency interconnection among nodes, leveraging RDMA, and since a container is not able to transparently use such network, information about what libraries and network drivers should be installed in a container will be provided. A subsection is dedicated to Intel Omni-Path network, together with some comparison performance tests about an application run on bare metal and within a container.

Since an HPC system can also be equipped to have accelerators, like GPUs, a section is dedicated to those specific cards with some performance benchmarking provided.

After the container deployment considerations, an integration with traditional batch job scheduler is explained. This mainly focuses on Slurm [9], providing details about options and flags, since it is currently the widely batch scheduler used in the HPC European infrastructures. Moreover, some possible optimizations are proposed.

In the last section, a particular case of cloud infrastructure is considered.

# 2 Application packaging in containers

To simplify the container building process, it is possible to use a package manager like Spack [4] or EasyBuild [5]. By using them, multiple versions, configurations, platforms, and compilers, and all of their builds can be chosen during the building process. By simply adding in the singularity recipe or in the docker file the specific commands of one of those package managers, the user can install the desired application. An example about how to use EasyBuild is reported in Deliverable D12.2 [6].

# 3 Containerization of Parallel MPI application

In this section, information about the Message Passing Interface (MPI) utilization within a container environment is provided. The usage of Open MPI [7] and Intel MPI [8] is considered in Singularity, as it is the containerization tool mostly used in HPC clusters.

## *3.1 Singularity and Open MPI*

Since the beginning, Singularity developers have spent time in the integration with Open MPI.
As reported in the admin guide [10], the Open MPI - Singularity workflow follows this scheme, also reported in Figure 1:

1. mpirun is called by the resource manager or by the user directly from a shell;
2. OpenMPI then calls the process management daemon (ORTED - the run-time layer or Open Run-Time Environment);
3. The ORTED process launches the Singularity container requested by the mpirun command;
4. Singularity builds the container and namespace environment;
5. Singularity then launches the MPI application within the container;
6. The MPI application launches and loads the OpenMPI libraries;
7. The Open MPI libraries connect back to the ORTED process via the Process Management Interface (PMI);
8. At this point the processes within the container run as they would normally directly on the host.

As written in [10], "this entire process happens behind the scenes, and from the users' perspective running via MPI is as simple as just calling mpirun on the host as they would normally".

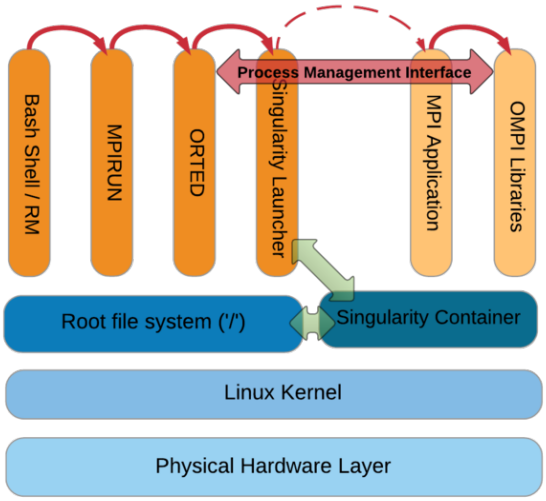


*Figure 1 - MPI Application process for singularity*
*http://www.admin-magazine.com/HPC/Articles/Singularity-A-Container-for-HPC*

To achieve proper containerized Open MPI support, a version of Open MPI greater or equal than 2.1 should be used.

Regarding to previous versions, Singularity developers underline that:

- Open MPI 1.10.x may work but it is expected an exact matching version of PMI and Open MPI on both host and container;
- Open MPI 2.1.0 has a bug affecting compilation of libraries for some interfaces, particularly Mellanox interfaces using libmxm are known to fail.

To create 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 OFED libraries should be installed into the container. As another option, it is possible to bind both Open MPI and networking libraries into the container, but this could run afoul of glib compatibility issues (it generally works if the container glibc is more recent than the host, but not the other way around). In section 5, details about the installation of Intel Omni-Path drivers are provided.

In the snapshot below, an example of recipe for the installation of Open MPI is given.

```
Bootstrap: docker
From: centos:latest
IncludeCmd: Yes

%post
yum -y groupinstall “Development tools”
yum -y install dapl dapl-utils ibacm infiniband-diags libibverbs libibverbs-
devel libibverbs-utils libmlx4 librdmacm librdmacm-utils mstflint opensm-libs
perftest qperf rdma

wget https://download.open-mpi.org/release/open-mpi/v2.2/openmpi-2.1.2.tar.gz
tar -xvf openmpi-2.1.2.tar.gz
cd openmpi-2.1.2
./configure --prefix=/usr/local/openmpi --disable-getpwuid --enable-orterun-
prefix-by-default
make
make install
```

Regarding the interoperability among different Open MPI versions, it is important to consider the role of Process Management Interface for Exascale Environments (PMIx).

As reported in [11], starting with v2.1.1, all versions are fully cross-compatible – i.e. the client and server versions can be any combination of release levels. Thus, a v2.1.1 client can connect to a v3.0.0 server, and vice versa.
PMIx v1.2.5 servers can only serve v1.2.x clients, but v1.2.5 clients can connect to v2.0.3, and v2.1.1 or higher servers. Similarly, v2.0.3 servers can only serve v2.0.x and v1.2.5 clients, but v2.0.3 clients can connect to v2.1.1 or higher servers.

Summarizing, the Open MPI/PMIx support best practices are:

| **OpenMPI version** | **Best practices** |
|---|---|
| Open MPI <= 1.X | Supports PMI, but not PMIx => Container and host OpenMPI/PMI versions must exactly match |
| 2.X <= Open MPI < 3.X | Supports PMIx 1.X |
| 3.X <= Open MPI < 4.X | Supports PMIx 1.X and 2.X |
| Open MPI >= 4.X | Supports also PMIx 3.X |

The general rule is that if the host Open MPI is linked with one of the latest PMIx, and the container supports PMIx (see PMIx compatibility matrix for more details), it will be compatible.

## *3.2 Singularity and Intel MPI*

Singularity developers have not spent much time in the usage of Intel MPI to run application within a container. Luckily, no particular issues seem to be noted in using Intel MPI.
Requirements to run an application in a singularity container using Intel MPI are:

1. Install infiniband libraries in the container
2. Make Intel MPI available to the container, by installing directly in it or binding the host directory with the `--bind` singularity option
3. Link the application against Intel MPI

Some compatibility tests have been performed on MARCONI HPC system [12], the CINECA Tier-0 supercomputing cluster, using Intel MPI parallel studio 2017 and Intel MPI parallel studio 2018 both in the host and in the container, and a perfect interoperability among such versions has been noted.

# 4 Networking libraries in a container

HPC infrastructures are characterized, among other things, to have an interconnection network inter-nodes featuring high bandwidth and low latency.
In order of a containerized application to take advantages from such networks, particular driver and libraries should be available in the container.
As cited in the previous section 3, the OFED libraries should be available in the container, to make it able to run over an Infiniband network. In what follows, the particular case of Intel Omni-Path Architecture is considered. All information is extrapolated from the application notes available at [13].

## *4.1 Intel Omni-Path Fabrics in a container*

The Intel Omni-Path Architecture (Intel OPA) offers high performance networking interconnect technology with low communications latency and high bandwidth characteristics ideally suited for HPC applications. The Intel OPA technology is designed to leverage the existing Linux RDMA kernel and networking stack interfaces. As such, many HPC applications designed to run on RDMA networks can run unmodified on compute nodes with Intel OPA network technology installed, benefiting from improved network performance. When these HPC applications are run in containers, these same Linux RDMA kernel device and networking stack interfaces can be selectively exposed to the containerized applications, enabling them to take advantage of the improved network performance of the Intel OPA technology.

The same standard OpenFabrics Enterprise Distribution (OFED)/OpenFabrics Alliance Software interfaces that are used when running over an Infiniband link layer (for example, as used for IPoIB, verbs, and RDMA) are also used when running over the OPA link layer; so, configuring to run containers on either is similar.

As reported in the Application Note available at [13], the basic steps to build a container able to use at the best Intel OPA are:

1. Install the latest Intel OPA-Basic release drivers and libraries from the Intel support site for your Linux distribution.

2. Decide which container technology is appropriate for your needs, install it, and run it.
   a. For Singularity, the Intel Omni-Path device is already available to a running container. Additional steps are not needed to use the Intel Omni-Path device interface.
   b. For Docker, add the Intel Omni-Path device hfi1_0in addition to the InfiniBand devices to the run line, as shown in the following example:

```
# ./docker run --net=host --device=/dev/infiniband/uverbs0 \
--device=/dev/infiniband/rdma_cm \ --device=/dev/hfi1_0 \ -t
-i centos /bin/bash
```

3. Install the application and any additional required user space libraries and software into a container image, and run it. User space libraries should include both the Infiniband user space libraries and the user space libraries needed to interface with the Intel Omni-Path driver.

For a simple recipe, Intel recommends to use "like on like", which means running on a standard supported OS distribution, such as CentOS 7.3, with a container image based on the same kernel and OS distribution, with a matching version of the Intel OPA-basic release used by each. Other combinations may work, but there is no support implied.

Here we provide an example of a Singularity recipe including the installation of Intel OPA driver. As always this has been reported in the Application Note available at [13]. In the same document, the reference to build a Docker container can be found.

```
Bootstrap: docker
From: centos:latest
IncludeCmd: yes

%post
yum -y install perl atlas libpsm2 infinipath-psm libibverbs qperf pciutils yum
-y install tcl tcsh expect sysfsutils librdmacm libibcm perftest rdma bc
yum -y install elfutils-libelf-developenssh-clients openssh-server
yum -y install libstdc++-devel gcc-fortran rpm-buildx compact-rdma-devel
yum -y install libibumad-devel libibumad-static libuuid-devel
yum -y install pci-utils which iproute net-tools
yum -y install libhfi1 opensm-libs numactl-libs


# Intel OPA driver installation #
cd /tmpdir
tar -xvf IntelOPA-Basic.RHEL73-x86_64.10.5.0.0.155.tgz
cd IntelOPA-Basic.RHEL73-x86_64.10.5.0.0.155

./INSTALL --user-space -n
```

### 4.1.1 Quantum Espresso singularity container test case on Cineca MARCONI system

In this section, performance results are provided in comparing the execution time of Quantum Espresso [14] in bare metal and within a singularity container.

The tests have been executed on MARCONI [12], 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.

Such supercomputer takes advantage of the Intel Omni-Path Architecture, which provides the high-performance interconnectivity 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 operative system is CentOS 7.3.1611. The software is available by modules, and the Singularity version used for testing was 3.0.1. Intel OPA driver compatible with those in MARCONI SkyLake node was installed in the containers.

Quantum Espresso is an integrated suite of Open-Source computer codes for electronic- structure calculations and materials modelling at the nanoscale. It is based on density- functional theory, plane waves, and pseudopotentials. The version of QE used is the 6.3, compiled with Intel parallel studio 2018 - update 4. The container has been built using Singularity 3.0.1.
In the containerized version, the Intel OPA driver have been also installed, with the same version that on MARCONI SkyLake where the tests have been made.

As tests, we used a small and a larger slab of the same material: the Zirconium Silicide, a mineral consisting of zirconium, and silicon atoms.

- small: 24 atoms in total with a K-point mesh of 6 x 19 x 13
- big: 108 atoms in total with a K-point mesh of 6 x 6 x 5

More details about this and other QE benchmarks can be found in [15].

The graphs below (Figure 2 and Figure 3) show the comparison between the application performance of bare metal vs container for both the above inputs. Total execution time are reported for the entire code (PWSCF) and for the Fastest Fourier Transform in the West (fftw), that is one of the most time-consuming part of the code. The calculations were repeated 5 times and then averaged on the set of data.
As it can be noted, the use of container doesn't introduce significant overhead, since the total execution time of bare metal and container runs are about the same.

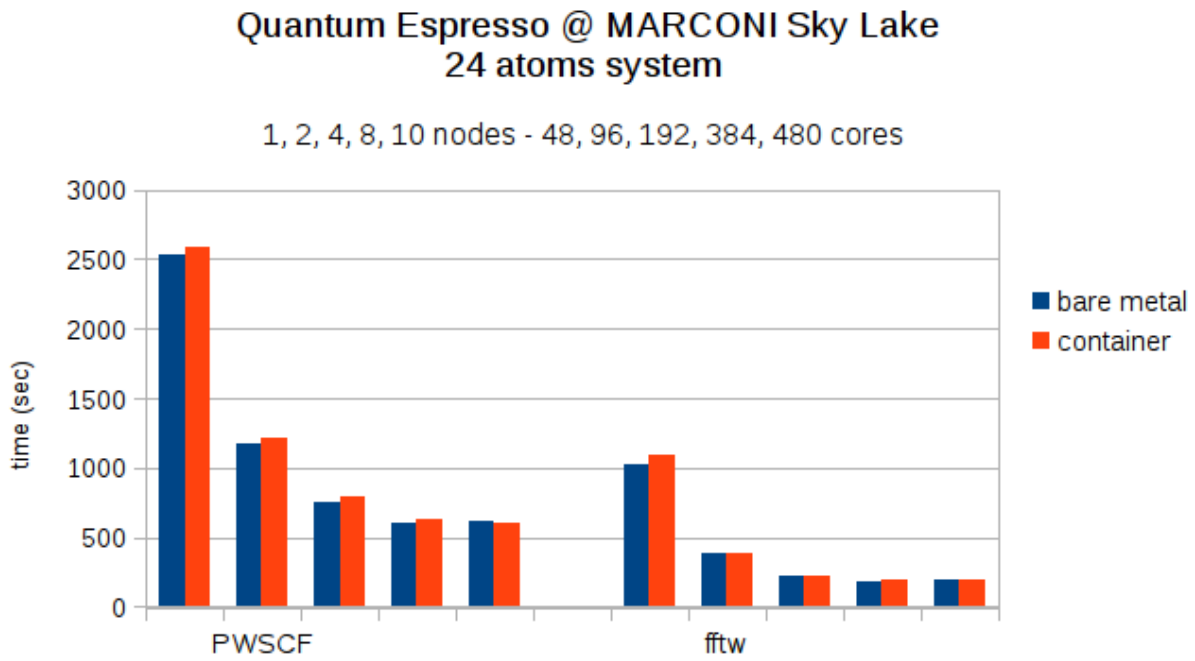


*Figure 2 - Total execution time of the simulation (PWSCF) and Fastest Fourier Transform in the West (fftw) are graphed for a system of 24 zirconium and silicon atoms run on 1, 2, 4, 8 and 10 MARCONI Sky Lake nodes. As reported, the overhead introduced by the use of containerization is not relevant.*

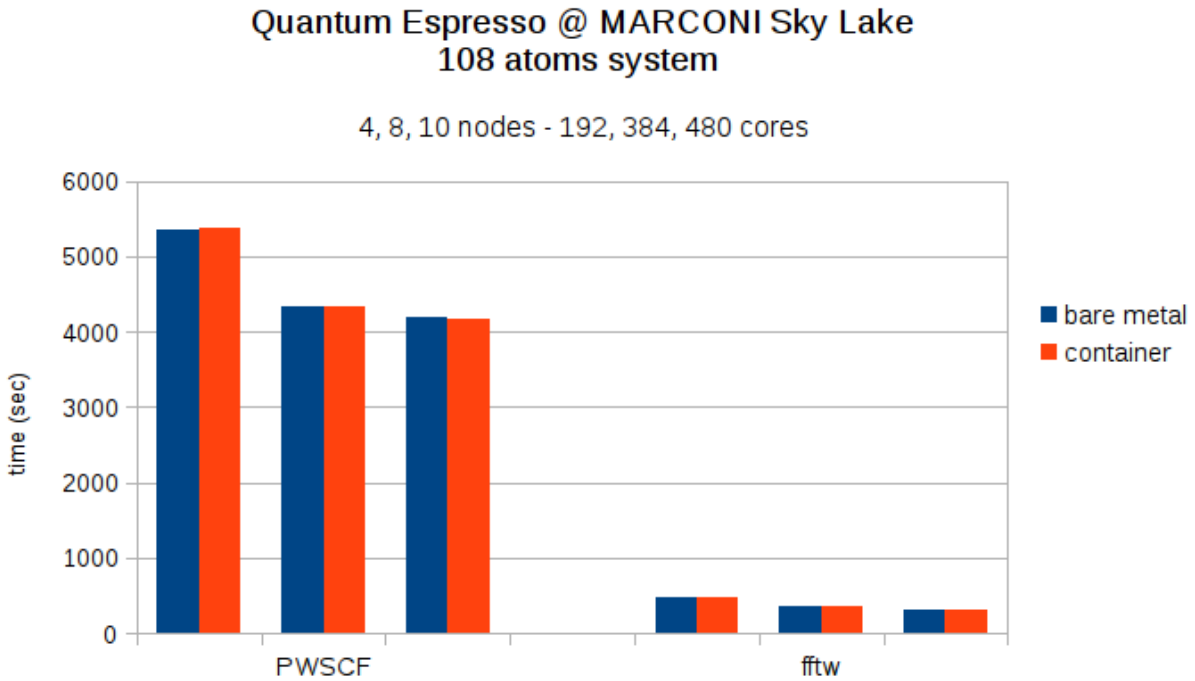


*Figure 3 - Total execution time of the simulation (PWSCF) and Fastest Fourier Transform in the West (fftw) are graphed for a system of 108 zirconium and silicon atoms run on 4, 8 and 10 MARCONI Sky Lake nodes. As reported, the overhead introduced by the use of containerization is not relevant.*

Another test is reported in Figure 4, in which Quantum Espresso version 6.4 has been compiled and run using OpenMPI 2.1.1 on MARCONI Sky Lake nodes. The same OpenMPI version was used in the host. The system simulated is a 24 zirconium and silicon atoms. As previously, the total execution time are reported for the entire code (PWSCF) and for the Fastest Fourier Transform in the West (fftw); the calculations were repeated 4 times and then averaged on the set of data. As it is possible to note from the graph, the overhead introduced using container is not relevant, since the total execution time of bare metal and container runs are about the same.

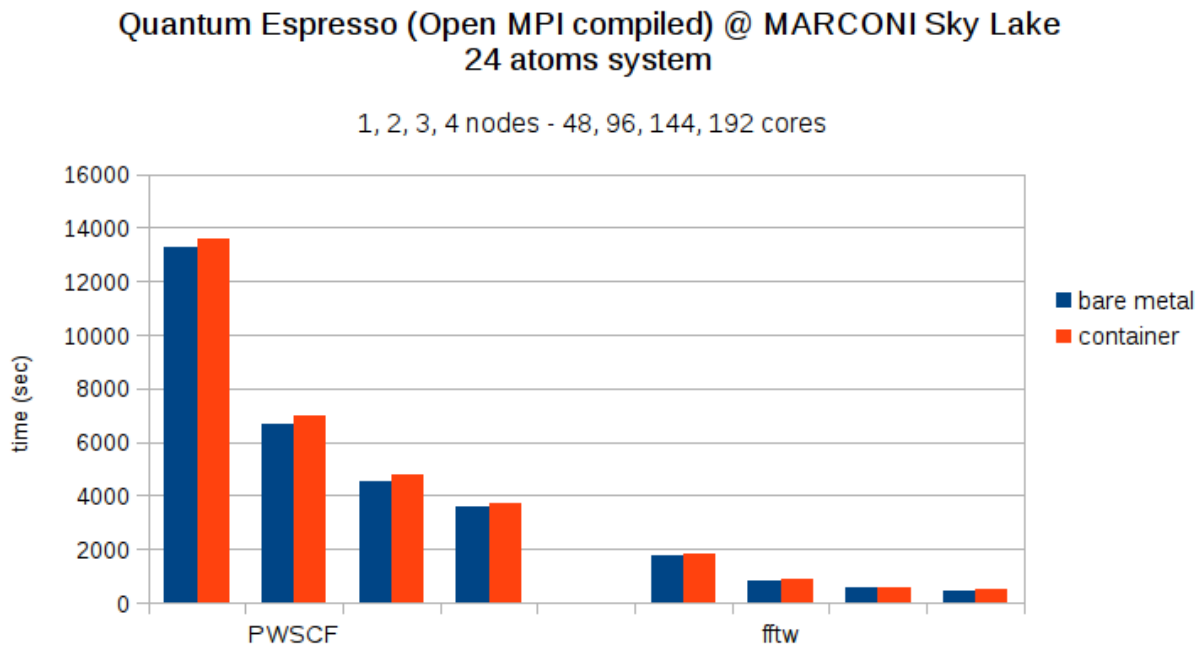


*Figure 4 - Total execution time of the simulation (PWSCF) and Fastest Fourier Transform in the West (fftw) are graphed for a system of 24 zirconium and silicon atoms run on 1, 2, 3 and 4 MARCONI Sky Lake nodes. In this case, the MPI library used was OpenMPI 2.1.1 both in the host and in the container. As reported, the overhead introduced by the use of containerization is not relevant.*

# 5 GPUs utilization within a container

Nowadays, many HPC systems have accelerators, as GPU, FPGA or Intel MIC. A container is by design platform-agnostic and also hardware-agnostic. This represents a problem when using specialized hardware such as NVIDIA GPUs which require kernel modules and user-level libraries to operate.
Of course, a solution is to install the GPUs drivers inside the container. But both Docker and Singularity provide other solutions. Singularity supports "natively" the GPUs in the sense that all the relevant NVIDIA/Cuda libraries on your host are found via the `ld.so.cache` that is binded into a library location within the container automatically. To enable the usage of the GPUs in a Singularity container, simply add the flag "`--nv`" in the singularity execution command line [2].
Docker developed a plugin, named NVIDIA-Docker; after the Docker installation and the plugin installation, simply run the container using the available wrapper "nvidia-docker" [16].

The Cuda Toolkit has to be available in the container, with a version compatible with the driver version installed on the GPUs in the host.

For other kind of accelerators, it is needed to have available the specific drivers and libraries into the container.

## *5.1 Tensorflow singularity container test case on Cineca GALILEO system*

In what follows, we report a test case of Tensorflow [17] application installed in a Singularity container run on GALILEO.

GALILEO [18] is the CINECA Tier-1 "National" level supercomputer. Introduced first in 2015, it has been reconfigured at the beginning of 2018 with Intel Xeon E5-2697 v4 (Broadwell) nodes. Also, available in GALILEO is a pool of 15 nodes, each equipped with 2 x 8-cores Intel Haswell 2.40 Ghz plus 2 nVidia K80 GPUs, 128 GB/node RAM, and an Infiniband with 4x QDR switches as Internal Network.

The container has been built directly from those available in the official Tensorflow Docker hub.
The version of Tensorflow used is the 1.12.0 for GPU. The container has been built using Singularity 3.0.2 version, the same also available as a module on GALILEO.
The recipe used to build the container is:

```
Bootstrap:docker
From:tensorflow/tensorflow:1.12.0-gpu-py3

%post
 mkdir -p /test
 chmod 777 -R /test
```

Both for bare metal and container tests, five different neural networks have been considered: AlexNet, GoogleNet, InceptionV3, ResNet-50, and VGG16. The dataset was ImageNet (synthetic), and three different batch sizes were analysed: 32, 64 and 128 per device, where the device is the GPU. All the runs have been executed on a single node with 2*8-cores Intel Xeon E5-2630 v3 @ 2.40GHz and 2 nVidia K80 GPUs.

The number of images per second is reported in the graphs below, figure 5, 6 and 7. The batch size being fixed, each histogram shows the number of images per second computed in each neural network model described above and for 1, 2, 3 and 4 K80 NVIDIA GPUs on a single GALILEO node. Note that the Model VGG16 and Inception V3 with a batch size of 128 are not shown because the run was out of memory. As shown, no overhead in the execution is introduced using the container instead of the bare metal application, since the number of images computed per second is more or less the same.

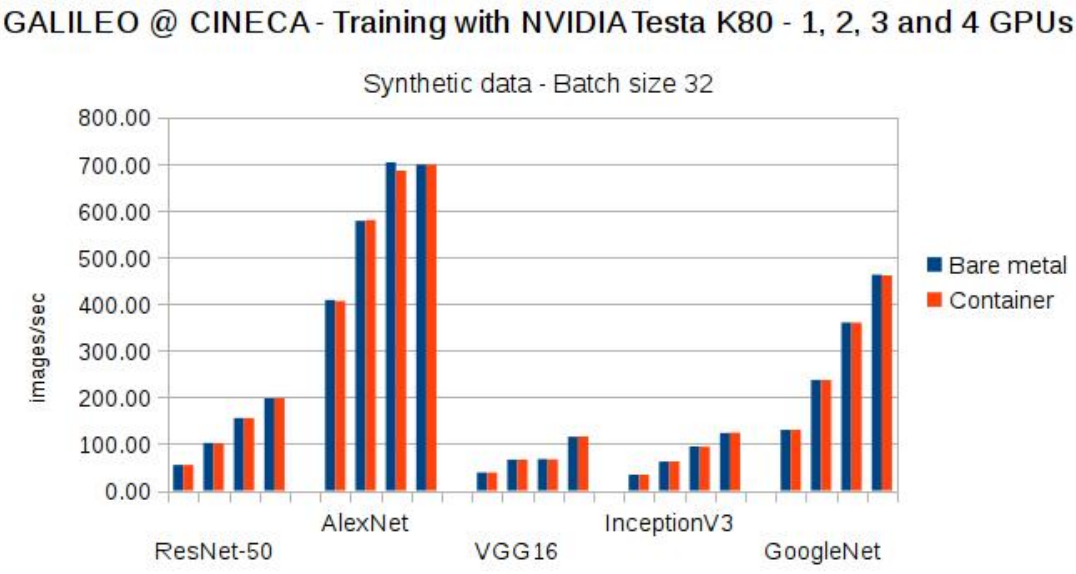


*Figure 5 - Number of images per second computed in ResNet-50, AlexNet, VGG16, InceptionV3 and GoogleNet model for 1, 2, 3 and 4 K80 NVIDIA GPUs on a single GALILEO node. The batch size used is 32, the dataset is ImageNet - synthetic*

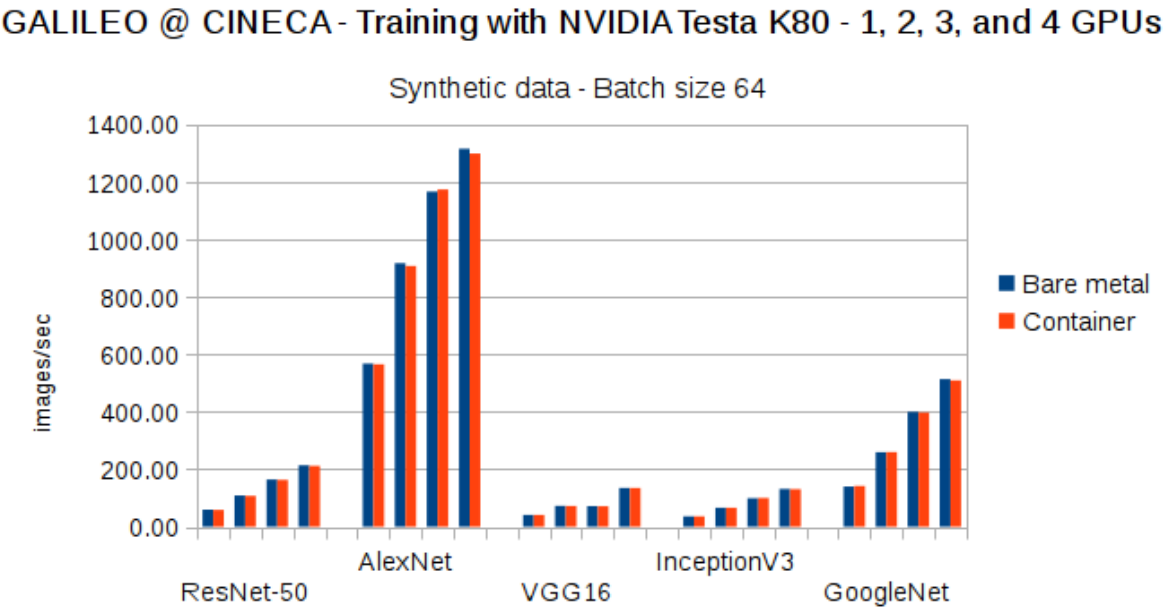


*Figure 6 - Number of images per second computed in ResNet-50, AlexNet, VGG16, InceptionV3 and GoogleNet model for 1, 2, 3 and 4 K80 NVIDIA GPUs on a single GALILEO node. The batch size used is 64, the dataset is ImageNet - synthetic*

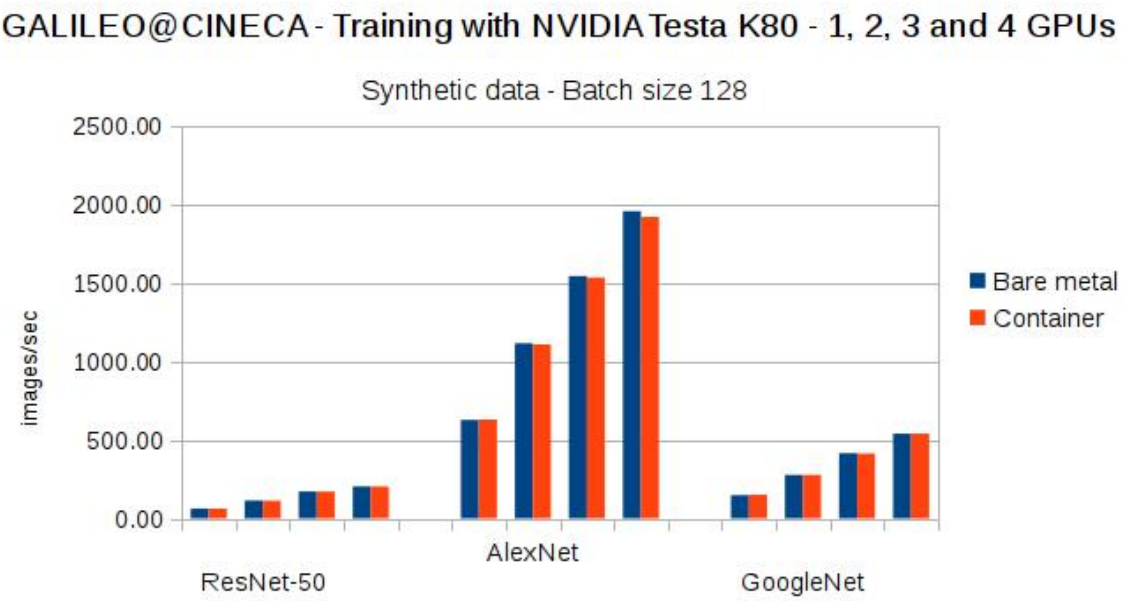


*Figure 7 - Number of images per second computed in ResNet-50, AlexNet, and GoogleNet model for 1, 2, 3 and 4 K80 NVIDIA GPUs on a single GALILEO node.*
*The batch size used is 128, the dataset is ImageNet - synthetic*

# 6 Integration with Batch Queuing Systems

Virtually all HPC systems use batch oriented workload managers to orchestrate the flow of submitted jobs through the system. Users create job specifications which are usually shell scripts with special directives to specify resources requested, job metadata, job constraints and dependencies, etc. as well as the actual application and its arguments to be run. By far the most commonly used workload manager at European HPC centres is Slurm [9], with Torque [19] and PBSPro [20] also reasonably common. While the syntax and back-end details are different for each type of workload manager, the core concepts and constituents of a job submission file are very similar from a users' perspective. Hence, we will use Slurm for the purposes of these discussions but most of the considerations here can be carried over to Torque, PBSPro, and others.

## *6.1 Integrating containers with workload managers*

The basic functionality of workload managers is to parse the resources request of a job (often simply a number of nodes for a specified time), prioritise that job relative to others and ultimately allocate the requested nodes to the job. The final step is to execute the job script supplied by the user, usually on the first node allocated to the job. Hence, the process by which an application is launched is not usually the responsibility of the workload manager - it just secures the requested nodes and then leaves it up to the user's script to manage the launching of the application.

A user job will typically choose one of three alternatives to run an application:

1. Simply launch a single executable which may be a serial process or parallel with the parallel task launching happening via the OS (such as with OpenMP applications) or encoded within the application (such as with some ISV applications that bundle their own MPI implementations and launchers).
2. For MPI applications, explicitly launch it using the mpirun or mpiexec task launcher which is part of the MPI distribution used to build the application. Slurm has inbuilt support in the srun command for launching MPI applications which support OpenMPI, PMI or ssh type launchers.
3. Workload managers also provide the ability to launch sub tasks within a job whereby multiple applications can be run in parallel or sequentially, each using a separate subset of resources. The Slurm command to do this is called srun and in Torque it is pbsdsh.

From an operating system and workload manager perspective, containers are very similar if not identical to normal application processes. Hence, Singularity, Charliecloud [21] and Shifter [22] can all be used in much the same way as normal applications and no special integration is required. For example, the following Slurm script illustrates a batch job which runs the WRF application inside a Singularity container:

```
#!/bin/sh
#SBATCH -N 1
#SBATCH -t 3:00:00
#SBATCH -A test

module load singularity
singularity exec /work/project1/singularity_images/eb-wrf.img wrf.exe
```

The script requests 1 node for 3 hours and charges the job to the test account. Once the job is started by Slurm, the script is run as a shell script and so the last 2 lines are executed in sequence to first load the singularity environment module (which sets PATH, etc.) and then run the actual singularity command as it would normally be done.
It is possible to simplify Singularity usage even further with Slurm through the use of a plugin which allows users to execute their Slurm jobs within a Singularity container without having to execute Singularity directly. To enable the plugin, add the following line to the Slurm plugin configuration (/etc/slurm/plugstack.conf):

```
required singularity.so
```

A user may then select the image through the --singularity-image optional argument and avoid calling singularity directly, instead simply specifying the application and its arguments:

```
#!/bin/sh
#SBATCH -N 1
#SBATCH -t 3:00:00
#SBATCH -A test
#SBATCH --singularity-image=/work/project1/singularity_images/eb-wrf.img

wrf.exe
```

## 6.2 Performance considerations with workload managers

As discussed above, workload managers' main function is in allocating resources and doing the initial launch of an application. They will not directly have an impact on the performance of an application and will treat bare metal and containerised applications equally in this regard. However, it is possible for workload managers to make specific changes to the operating system environment and settings for jobs launched under their management. In particular, workload managers can control how processes/tasks are distributed across multiple nodes within a job and within nodes can use underlying OS features like cpusets and cgroups to control how memory and CPU resources are allocated and the mapping of physical cores to processes. These features will work in the same manner for containers as for normal applications.

Slurm provides a very comprehensive set of plugins and configuration options to help administrators and users to control process placement. The interaction of so many configuration options can lead to unexpected results and so careful testing and verification is required to ensure that the intended behaviour is actually happening in reality. Some of the more commonly used command options and configuration settings are described below to give an overview of the capabilities. The official documentation sources should be consulted for up to date, detailed information.

| **`slurm.conf` Configuration Option** | |
|---|---|
| TaskPlugin=task/affinity | enables resource containment using CPUSETs. This enables the --cpu-bind and/or --mem-bind srun options. |
| TaskPlugin=task/cgroup | enables resource containment using Linux control cgroups. This enables the --cpu-bind and/or --mem-bind srun options. |
| **`srun` Task invocation control** | |
| --cpus-per-task=CPUs | number of CPUs required per task |
| --ntasks-per-node=ntasks | number of tasks to invoke on each node |
| --ntasks-per-socket=ntasks | number of tasks to invoke on each socket |
| --ntasks-per-core=ntasks | number of tasks to invoke on each core |
| --overcommit | Permit more than one task per CPU |
| -m / --distribution | Specify alternate distribution methods for remote processes. |
| **`srun` Application Hints** | |
| --hint=compute_bound | use all cores in each socket |
| --hint=memory_bound | use only one core in each socket |
| --hint=[no]multithread | [don't] use extra threads with in-core multi-threading |
| **`srun` Low level affinity binding** | |
| --cpu-bind | Fine grained control and reporting of how tasks are mapped to physical CPUs |

# 7 Optimizations identified

## *7.1 Singularity integration with MPI*

MPI integration is crucial to every HPC container as it involves portability and performance issues. Portability is affected by the broad hardware and software implementations of MPI. It is known that every HPC centre possesses a different network interconnection with a custom MPI deployment. Moreover, many centres take advantage of high-performance networks, e.g. Mellanox or Infiniband implementations, which help achieve performance boosts but are not generic to all clusters.

Though Singularity may leverage host's MPI configuration in further releases as done with Nvidia support, currently users or system administrators have to deal with this issue. The most generic and easiest approach to enable MPI communications among containers is to install within the container the same MPI software and version as in the host. One should be able to launch their Singularity container using MPI without problems, however, they will only be able to use generic byte transfer layers as TCP. For instance, if the cluster where the container is running has Infiniband, it will not detect it and it might affect negatively its applications performance. A workaround for high-performance networks detection issue is to directly apply host MPI installation inside the container image and everything it can involve: driver updates, kernels modules, configuration files, etc. As a result, containers would have the same MPI configuration as the host and therefore there would not be problems to leverage high-performance networks.

Nevertheless, rebuilding Singularity images from scratch with the proper MPI installation for every cluster would badly affect the portability. A better solution exists: binding the entire host MPI related files within the container at launch time. To do so, one has to use the user-bind option (flag -B) or specifying the paths in Singularity's configuration file so it is automatically done without user's intervention.

For example, imagine that we have a cluster with Open MPI 1.10.4 installed in "/apps/mpi/openmpi-1.10.4" with some dependencies in "/lib64". Besides, the cluster disposes of an Infiniband network with its files located in "/usr/infiniband/" and a configuration file in "/etc/libibverbs.d". In this scenario, we could copy the necessary dependencies from "/lib64" to "my_container_lib64" folder in our home to avoid overwriting container's "/lib64" with the host. As "/apps/mpi/openmpi-1.10.4" path should not exist in our container directory, the same for "/usr/infiniband", we can directly bind those paths as follows:

```
-B /apps/mpi/openmpi-1.10.4:/apps/mpi/openmpi-1-10-4 _B
```

Last but not least, it is necessary to adapt PATH and LD_LIBRARY container variables so its system is able to find the proper binaries and libraries.

# 8 Application containerization using cloud resources

We experimented with various ways of running containers: directly on HPC system using MARCONI and GALILEO platforms provided by CINECA, on the CSC IaaS cloud platform cPouta [23] and through Kubernetes/OpenShift based container orchestration service Rahti [24].

Singularity containers are highly portable. The main challenges are related to MPI integration and GPU driver stack as discussed in sections 3 and 5 respectively. In most cases, it is preferable to run them directly on the HPC platform. The use of container can even be made, if so desired, invisible to end users by use of suitable wrapper scripts. From maintenance side, it is often preferable to manage all resources through a single batch job system.

There can be cases where this is not technically possible or is undesirable for some other reason. For example, the original plan was to implement Singularity on CSC Taito platform [25]. It was found out that Taito operating system, especially the kernel version, was too old to support OverlayFS, thus making it impossible to mount host file systems from a container. Update of system was not deemed feasible as Taito was nearing the end of its life.

While Singularity containers can be run with user level rights, building containers requires root access. Virtual machines can provide suitable development platform that allow users to build and test their containers before deploying them in to HPC platform.

IaaS style cloud platform, where users start a virtual machine to run their containers, provides some advantages, but it has also many limitations.

The main advantage is compatibility. Host operating system can be changed much more freely and adjusted compared to big HPC systems where overall compatibility and stability is the main concern.

A dedicated VM makes it also less problematic to run programs with root privileges, making running Docker containers feasible. This is helpful e.g. in cases where Docker containers provided by software developers are not directly compatible with Singularity. This has become less of an issue as Singularity can use a Docker image to build a Singularity image.

Dedicated VM will naturally also facilitate direct software installations, but containers can still be a preferable way of software distribution, especially in cases where the software stack is complex.

The main limitations are related to scalability. Clouds work well with single-node applications. They are also suitable for scenarios where simple tasks are distributed among several independent compute nodes. They are not, however, optimal for MPI heavy tasks, where interconnect speed and topography can lead to major performance bottlenecks.

There can also be complications on using dedicated hardware assets such as GPUs and SSDs. While they can be made available for cloud VMs, it is often undesirable from resource allocation point of view. In most systems, these are very limited resources and it is usually more flexible to manage them directly through workload managers.

One limitation is also the additional skill requirements for users. The end users are often scientists with limited IT background and setting up their own servers through the IaaS system can be a daunting task, even when VM images with required software stacks are provided. This can be circumvented with container orchestration systems, which simplify container deployment from the end user's point of view.

## *8.1 Utilizing containers in Elmer development and production – Case study*

Currently, containers are used in Elmer software [26] production solely for testing using travis-ci.org platform. We are investigating in building Elmer deb/rpm/tgz binary packages using containers and providing a container image with Elmer to end users. Choosing containers to do these tasks was reasonable because the image build recipes are easily transferable and modifiable for other purposes than testing/packaging as well. A particular challenge in the case of Elmer, is that it depends on a large number of external libraries, which are not shipped with the source code and their installation is different depending on end user operating system. For example, compiling Elmer on Mac OS is fragile and providing a working recipe requires frequent tweaking.

With containers, it is easy for users to have a specific version of Elmer compiled without ever having to issue a single build command apart from "docker build". This could be an important case where users want to have a standardized simulation environment, for example.

Containers allow the building of tailored, possibly highly customized, simulation workflows on top of a bare image containing just the Elmer solver. Such containers, again, standardize the user's simulation environment and testing of new features in the workflow is easy.

An Elmer container with a tailored workflow can also be seen as a commercial product and, given that the user interface is sufficiently polished, the hypothetical Elmer container could be embedded as a part of a simulation suite in a company.

Typically, container platforms don't allow bare-metal access to fast interconnect but there are finite element simulations that don't suffer dramatically from a virtualization overhead in message passing. There are real use cases for Elmer that utilize only 4-32 cores and 8GB-64GB of memory, both requirements of which are easily satisfied on container platforms. Computational jobs of this scale are not infrequent also among HPC-Europa3 visitor projects. Containerization and cloud usage might be the path of least resistance for an application which is hard to compile e.g. on the rather old OS available at the current CSC HPC platform.

Another good reason to utilize docker containers is that the same Docker files can be utilized in HPC settings using singularity containers.

# 9 Conclusion

In this deliverable, it has been discussed how container technologies can be used to improve portability of application in HPC has been discussed. Firstly, an investigation about containers building has been carried out. Considerations about the availability of parallel library, network library and accelerators within a container have been presented.

To obtain an optimised use of the hardware available in an HPC system, the network driver libraries and most suitable compiler should be available in the container, either by installing them inside or making them available by binding host library directories. However, to build a portable container, it is sufficient that open MPI and basic infiniband network library are present in the container. In any case, by design, MPI library must be present in the host, with some constraints about versions and distribution to avoid compatibility issues.

Rules to use GPUs within a container have been also reported, together with a test case performance benchmark. As Singularity is able to automatically bind GPUs host library, the user doesn't have to install any driver. It has only to take care that the cuda toolkit version used was compatible with the GPU driver version installed in the host.

After those building hints, the integration with traditional batch schedulers has been considered, focusing on SLURM since it is the workload manager available in all the HPC-Europa3 sites. An example of job batch script has been reported together with the possible configuration options actually available in order to configure SLURM at the best for using containers.

The proposed optimization regards the possibility to bind host libraries within a container, to avoid, as an example, the installation of certain drivers during the container building process. By such binding, in an easy and transparent way, a user should have the host library available in the container.

Finally, a nice use case of containerization in a cloud environment was presented, citing as use case Elmer code that is frequently used by HPC-Europa3 visitors at CSC.

# References


1. N. Wilson, R. Sirvent, M. Renato, O. Rudyy, G. Muscianisi, Y. Cardenas, R. Laurikainen, A-M. Saren, D. Dellis, "Container-as-a-service analysis report," D12.1, 2018 [Online]. Available: http://doi.org/10.23728/b2share.c93bf40018f74f04ab8db4636f55f143.
2. Singularity: https://www.sylabs.io/docs/
3. Docker: https://www.docker.com/
4. Spack: https://spack.io/
5. Easybuild: https://easybuild.readthedocs.io/en/latest/
6. N. Wilson, O. Rudyy, A. Sillanpää, "Container-as-a-service Technical Documentation" D12.2, 2019. Under review.
7. OpenMPI: https://www.open-mpi.org/
8. Intel MPI: https://software.intel.com/en-us/mpi-library
9. Slurm: https://slurm.schedmd.com/documentation.html
10. Singularity Admin Guide for OpenMPI: https://www.sylabs.io/guides/2.6/Admin-guide.pdf%20
11. "How does PMIx work with containers?" [Online] https://pmix.org/support/faq/how-does-pmix-work-with-containers/
12. MARCONI user documentation: https://wiki.u-gov.it/confluence/display/SCAIUS/UG3.1%3A+MARCONI+UserGuide
13. "Building Containers for Intel Omni-Path Fabrics using Docker* and Singularity*" [Online] 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.
14. Quantum Espresso: https://www.quantum-espresso.org/
15. Quantum Espresso benchmark: https://github.com/electronic-structure/benchmarks
16. NVIDIA-Docker GitHub: https://github.com/NVIDIA/nvidia-docker
17. Tensorflow: https://www.tensorflow.org/
18. GALILEO user documentation: https://wiki.u-gov.it/confluence/display/SCAIUS/UG3.3%3A+GALILEO+UserGuide
19. Torque: http://www.adaptivecomputing.com/products/torque/
20. PBS Pro: https://www.pbspro.org/
21. Charliecloud: https://github.com/hpc/charliecloud
22. Shifter: https://github.com/NERSC/shifter

23. cPouta user documentation: https://research.csc.fi/pouta-user-guide

24. Rahti user documentation: https://rahti.csc.fi/

25. Taito user documentation: https://research.csc.fi/taito-user-guide

26. Elmer: finite element multiphysics simulation software developed at CSC. https://www.csc.fi/fi/web/elmer