An efficient, portable and flexible container solution for fast deployment in an HPC infrastructure

Manuel Simon Novoa

September 1, 2017

Contents

1 Introduction

Throughout this document I will describe the study on an **efficient, portable and flexible container solution** for science reproducibility and HPC purposes. This research is composed by several activities that will be done under the HPC infrastructure of the ***Supercomputing Center of Galicia (CESGA)*** as a member of the European collaborative project *"Mathematical Modeling, Simulation and Optimization for Societal Challenges with Scientific Computing (**MSO4SC**)"*[1]. This project is funded by the European Commission through the E-INFRA ***Horizon 2020 Program***[3], the European Union's largest research and innovation program.

In particular, the work explained throughout this document presents the motivation why Singularity, a containerization technology, is selected, as well as a research study in order to obtain a general enough work-flow and configuration for running this solution in an HPC environment. Thus, I will describe several approaches in order to obtain portability and flexibility for this container solution while running parallel applications, its configuration in the HPC infrastructure of the center and the realization of multiple performance tests to ensure the usage of these HPC resources.

2 Project overview

This document, as well as all tests and obtained conclusions belong to the european collaborative project *"Mathematical Modelling, Simulation and Optimization for Societal Challenges with Scientific Computing" (MSO4SC)*. A project overview could be: provide mathematical technologies and its applications for solution of societal challenges as a service through an HPC oriented cloud e-infrastructure. Going into more detail, the project is defined as:

“The challenges of society show rising complexity and their solution process increasingly requires a holistic approach. It is necessary to provide decision makers with tools that allow long-term risk analysis, improvements or even optimization and control. One of the key technologies in this process is the use of mathematical Modelling, Simulation and Optimisation (MSO) methods, which have proven effective tools for solving many problems, e.g. realistic prediction of wind fields, solar radiation, air pollution and forest fires, prediction of climate change, improving the filtration process for drinking water treatment and optimization methods for intensity-modulated radiation therapy.

These methods are highly complex and are typically processed via the most modern tools of ICT including high performance computing and access to big data bases and usually need support of skilled experts, which often are not available, in particular in small and medium enterprises.

To improve this situation, the pan-European network EU-MATHS-IN (European Service Network of Mathematics for Industry and Innovation) has been founded from national networks containing leading research centres of Europe with high excellence of MSO and ICT.

The major objective of this proposal is to construct an e-infrastructure that provides, in a user-driven, integrative way, tailored access to the necessary services, resources and even tools for the fast prototyping, providing the service producers with the mathematical frameworks as well.

The e-infrastructure consists of an integrated MSO application catalogue containing models, software, validation and benchmark and the MSOcloud: a user friendly cloud infrastructure for selected MSO applications and developing frameworks from the catalogue. This will reduce the 'time-to-market' for consultants working in the above-mentioned societal challenges."

3 HPC architecture description

The tests and executions named in the following document were launched it all under the hardware described in this section. As this research belong to the *Supercomputing Center of Galicia (CESGA)*, the selected machine to perform the development was the *Finis Terrae II*.

Finis Terrae is a scientific computing infrastructure managed by *CESGA* and integrated in RES (the Spanish Supercomputing Network), a Spanish Unique Scientific and Technical Installation (ICTS). It is a linux based heterogeneous cluster, with an InfiniBand FDR low latency network interconnecting 317 computing nodes based on Intel Xeon Hasswell processors. Each node is composed by 24 cores and has a 128GB main memory per server. The cache structure of each core is the following one:

- L1d cache: 32K
- L1i cache: 32K
- L2 cache: 256K
- L3 cache: 30720K

The cluster is connected to a shared Lustre High-performance Filesystem with 768TB of disk space. Additionally, the system has 4 GPU servers with GPUs (NVIDIA K80) and 2 servers with Intel Xeon Phi accelerators. There is also one "Fat" node with 8 Intel Haswell 8867v3 processors, 128 cores and 4TB of main memory. Together, these nodes are able to provide a computing power of 328 TFLOPS, 44,8 TB of RAM and 1,5 PB of disk capacity. The system includes a high performance parallel storage system able to achieve a speed of more than 20 GB/s.

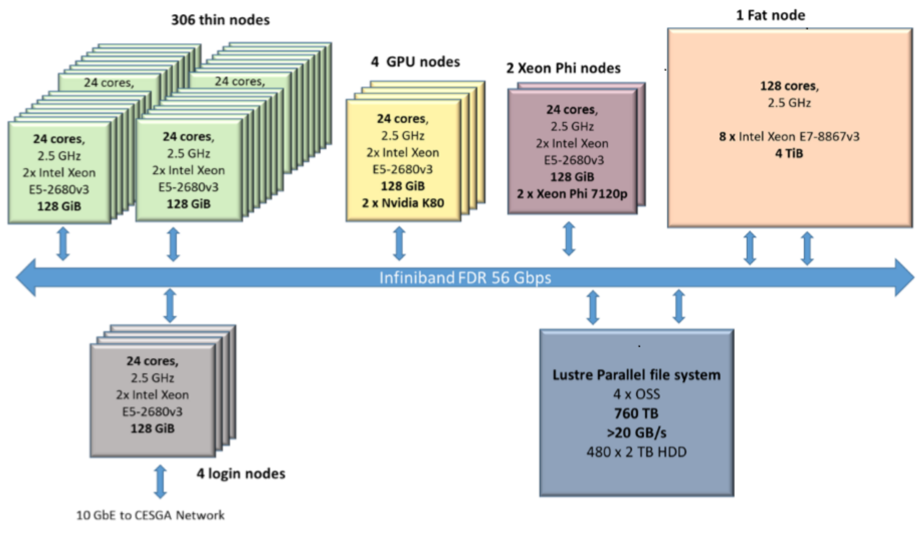
306 thin nodes
24 cores, 2.5 GHz 2x Intel Xeon E5-2680v3 128 GiB
4 GPU nodes
24 cores, 2.5 GHz 2x Intel Xeon E5-2680v3 128 GiB 2 x Nvidia K80
2 Xeon Phi nodes
24 cores, 2.5 GHz 2x Intel Xeon E5-2680v3 128 GiB 2 x Xeon Phi 7120p
1 Fat node
128 cores, 2.5 GHz 8 x Intel Xeon E7-8867v3 4 TiB
Infiniband FDR 56 Gbps
24 cores, 2.5 GHz 2x Intel Xeon E5-2680v3 128 GiB
4 login nodes
10 GbE to CESGA Network
Lustre Parallel file system 4 x OSS 760 TB >20 GB/s 480 x 2 TB HDD

Figure 1: Finis Terrae II Infrastructure

4 Research motivation

Traditional software deployment performed in HPCs presents some issues which make it very inflexible. Different approaches, like could be the containerization technology, 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 this task is to provide the 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. Lmod, a Lua based module system, is being used at Finis Terrae II.

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 run a particular application, but to manage complex work-flows with environment modules can be sometimes unaffordable, 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 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.

So, 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.

5 Technology selection

In accordance with the requirements mentioned in the previous section, a study of the state of the art was made to select the solution that best fits the presented problem.

It is very important to always keep in mind these premises, since they are the ones that will limit the selection of the solution:

- The operating system and kernel of the cluster will not be modified during this research.
- We must make use of the available hardware (shown in section 3) as efficiently as possible; that is, we can not limit the resources (CPU, memory, network, etc.)

Thus, some common virtualization solutions are:

5.1 Emulation

An emulator typically functions as an interpreter who reads the instructions from the virtual operating system (guest) and translates them into instructions that can run smoothly on the host operating system. In this way the emulator provides the virtual operating system with a simulated execution environment. The emulator itself runs like any other application on the host operating system. The emulated hardware may or may not match the actual machine on which the emulator is running. Emulators can recreate a complete environment, including all CPU instructions and I/O devices using software, which is why performance is deeply affected.

In addition, the use of emulators implies by definition the limitation of the resources available to each machine. That is, whenever we define a virtual machine, we must assign to it a series of resources such as the number of processors, the size of RAM to be used, etc. These resources will be unique to each machine, making a complicated balancing of available resources in the cluster. That is, once a machine is created with certain associated resources, these resources will be exclusive and appropriative, so if the machine does not use them to their fullest (something certainly complicated that happens at all times), resources will be underutilized.

So, as many of the scientific software that will be employed under the future deployment solution will make intensive use of machine resources, emulation solution does not seem to be the best. In order to respect the second premise, this solution will be discarded.

5.2 Virtualization: Hosted approach and paravirtualization

In hosted approach, the virtualization layer runs on a host operating system, and virtualized servers run on it. The paravirtualization allows the use of different operating systems, but it has the great disadvantage that the entire I/O passes through all the layers, causing all of them to absorb resources and causing this I/O to become very slow. Therefore, its use is not recommended for applications with intensive I/O.

In order to avoid this problem, some hosted approach products are evolving towards the paravirtualization of the I/O. In paravirtualization, we use operating systems that have been slightly modified to run in a virtualized environment, in order to replace all problematic operations. The guest operating system has also been modified and uses a special API to communicate with the virtualization layer and have relatively direct access to the hardware. The purpose of the modifications is to achieve the kernel of the virtual operating system to cooperate with the kernel of the host operating system. In this case, the guest operating system is modified to use an ABI (Application Binary Interface).

An example of paravirtualization might be KVM (Kernel-based Virtual Machine) or the Xen hypervisor. To clarify all terms, a hypervisor is a virtualization software, a thin layer of software that runs directly on the hardware, multiplexes the accesses of the different operating systems to the hardware, in addition to providing an environment for its control and management (the domain 0). It does not need a host operating system and under this layer virtual machines are created. The guest operating systems run on these virtual machines. Because of this, hypervisors are lighter and more efficient than the hosted approach.

Currently, the evolution of Xen and KVM is to minimize the overhead created by the hypervisor and allow the virtual machines to have almost direct access to the hardware.

Finally, we must take into account that all these peculiarities of emulation and virtualization will lead to a rather slow deployment, due to the creation of the machine and allocation of own resources in each case. As we require a fast deployment solution under HPC infrastructures, this slow starting up is unacceptable and the main reason why these solutions are rejected for our purpose.

5.3 Containerization: Operating System Virtualization

This type of virtualization runs only a kernel, the host, which also attends calls from virtual systems. In general, containerization software uses a set of libraries that translate or redirect virtual operating system calls into the appropriate format to be executed on the hardware using the host kernel. The environment provided by the container may or may not match the actual machine on which it is run. In case it does not match, the translator will be quite complex.

Each container runs in a private and contained environment, although they share the amount of resources of the host without the need of allocation, which is a benefit comparing with virtualization. This means that same hardware can host more containers than virtual machines, although it is possible to reallocate container resources to implement quality of service.

Therefore, this is a solution that allows flexibility in addition to a minimum overhead.

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 key feature for the project. 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. There is a stiff competition to push different technologies in the market. Although Docker is the most popular, to choose the right technology, depending on the purpose, it is important to understand what each of them stands for and does.

Several containerization technologies (like LXC, Docker, Udocker and Singularity) have been tested in the context of this project, but finally, Docker was rejected because of its kernel requirements and security. For example Finis Terrae II does not have the kernel required by Docker. 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.

In one hand, Udocker 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.

Although the Udocker development team is working to integrate it with message passing interface libraries (MPI), unfortunately, it is not yet supported.

On the other hand, 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 shows the work-flow for creating and running containers using Singularity at Finis Terrae II.

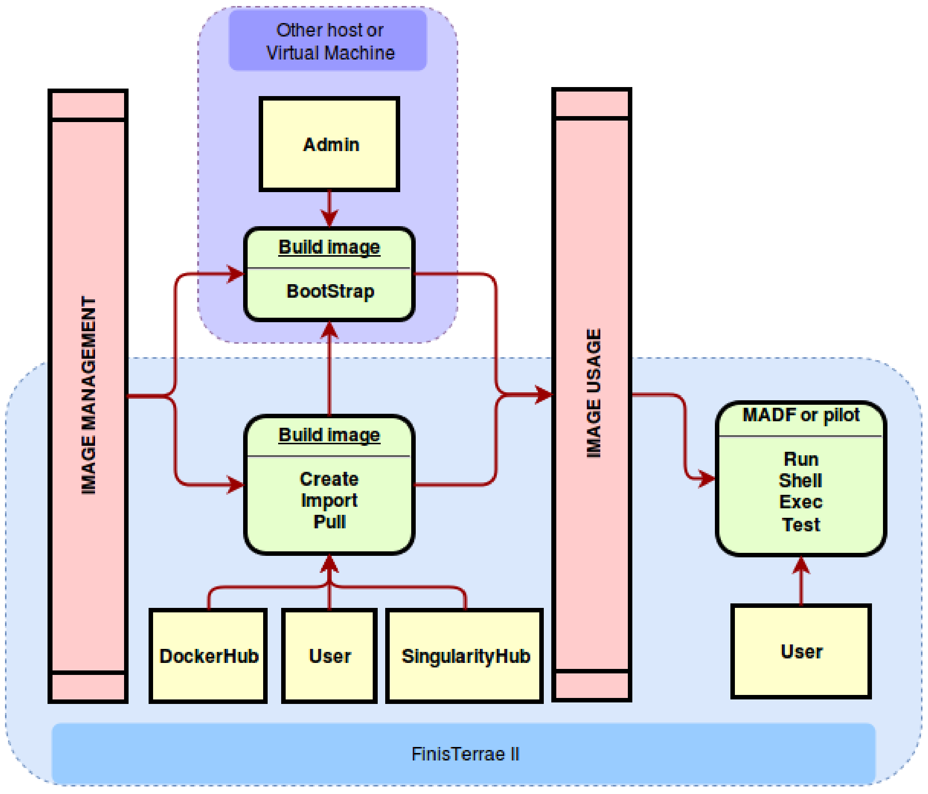

Figure 2: Singularity work-flow at Finis Terrae II

As we can see in the previous figure, users can create or pull images from public registries (like Docker Hub[5] or Singularity Hub[6]), and also import images from tar pipes. Once the image is created, Singularity allows to execute 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 Finis Terrae II, except the bootstrap process that needs to be called by a superuser. 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.

6 Involved technologies

The hardware under all tests and executions were executed were already described in section 3. Now it is turn to describe all the involved software needed to perform the solution, as well as all the implications they could have.

6.1 Host Operating System and kernel

The Finis Terrae II host machine have a ***Red Hat Enterprise Linux Server (RHEL) release 6.7 (Santiago)***, with a kernel version `Linux 2.6.32-573.12.1.el6.x86_64`. Fortunately, this kernel version is compatible with Singularity.

6.2 Singularity

In order to understand Singularity's operation mode, some explanations will be done:

6.2.1 Process Flow

The goal of Singularity is to run an application within a contained environment such as it was not contained. Thus there is a balance between what to separate and what not to separate. At present the virtualized namespaces are process, mount points, and certain parts of the contained file system.

On the other hand, Singularity does not support user escalation or context changes, nor does it have a root owned daemon process managing the container namespaces. It also exec's the process work-flow inside the container and seamlessly redirects all I/O in and out of the container directly between the environments. This makes doing things like MPI, X11 forwarding, and other kinds of work tasks trivial for Singularity.

When executing container commands, the Singularity process flow can be generalized as follows:

1. Singularity application is invoked
2. Global options are parsed and activated
3. The Singularity command (subcommand) process is activated
4. Subcommand options are parsed
5. The appropriate sanity checks are made
6. Environment variables are set
7. The Singularity Execution binary is called (sexec)
8. Sexec determines if it is running privileged and calls the SUID code if necessary
9. Namespaces are created depending on configuration and process requirements
10. The Singularity image is checked, parsed, and mounted in the CLONE_NEWNS namespace

11. Bind mount points are setup

12. The namespace CLONE_FS is used to virtualize a new root file system Singularity calls `execvp()` and Singularity process itself is replaced by the process inside the container

13. When the process inside the container exists, all namespaces collapse with that process, leaving a clean system

All of the above steps take approximately 15-25 thousandths of a second to run, which is fast enough to seem instantaneous.

6.2.2 Images

Singularity images are single files which physically contains the container. The effect of all files existing virtually within a single image greatly simplifies sharing, copying, branching, and other management tasks.

You do not need admin/sudo to use Singularity containers. You do however need admin/root access to install Singularity and to build/manage your containers and images, but to use the containers you do not need any additional privileges to run programs within it.

Because Singularity is based on container principals, when an application is run from within a Singularity container its default view of the file system is different from how it is on the host system. This is what allows the environment to be portable. This means that root (/) inside the container is different from the host!

Singularity automatically tries to resolve directory mounts such that things will just work and be portable with whatever environment you are running on. This means that /tmp and /var/tmp are automatically shared into the container as is /home. Additionally, if you are in a current directory that is not a system directory, Singularity will also try to bind that to your container.

Note that depending on kernel version, if Singularity version is below 2.3.1, there is a caveat in that a directory must already exist within your container to serve as a mount point. If that directory does not exist, Singularity will not create it for you! You must do that.

6.2.3 Resources sharing

Singularity does no network isolation because it is designed to run like any other application on the system. It has all of the same networking privileges as any program running as that user.

Singularity also allows you to leverage the resources of whatever host you are on. This includes HPC interconnects, resource managers, file systems, GPUs and/or accelerators, etc. Singularity does this by enabling several key facets:

- Encapsulation of the environment
- Containers are image based
- No user contextual changes or root escalation allowed
- No root owned daemon processes

6.2.4 Bootstrap

Bootstrapping is the process where we install an operating system and then configure it appropriately for a specified need. To do this we use a bootstrap definition file which is a recipe of how to specifically build the container.

The bootstrap command is useful for creating a new bootstrap or modifying an existing one using a definition file that describes how to build the container.

A Bootstrap definition file contains two main parts:

1. **Header**: The core operating system to bootstrap whithin the container.

   The Bootstrap: keyword Identifies the singularity module (yum, debootstrap, arch, docker) that will be used for building the core components of the OS. Depending on the loaded module several keywords can be used to particularize the installation (MirrorURL, OSVersion, Include, From, etc.)

2. **Sections**: Shell scriptlets to run during the bootstrap process. The Boostrap sections are:

   - setup: A Bourne shell scriptlet which will be executed on the host outside the container during bootstrap.
   - post: A Bourne shell scriptlet executed during bootstraping from inside the container.
   - runscript: A persistent scriptlet in the container that will be executed via the singularity run command.
   - test: A persistent scriplet in the container that will be executed via the singularity test command. This section will be also executed at the end of the bootstrap process.

Since Singularity operation should already be clear, only the Singularity versions used can be indicated: In the **local machine**, where root permissions are available, the Singularity version employed is **2.3-dist**. The Singularity version available on the **cluster** is **Singularity 2.2.1**.

6.3 MPI

6.3.1 Libraries

Since it is intended to use MPI, libraries will be used to enable this task. The selection of these libraries was done just checking those already installed on the cluster. The main efforts were put in the properly work of Open MPI, but some tests were done with Intel MPI too. This is the list with the MPI libraries and versions which are available on the cluster:

- Open MPI
  - Open MPI 1.10.2
  - Open MPI 2.0.0
  - Open MPI 2.0.1
  - Open MPI 2.0.2

  - Open MPI 2.1.1

- Intel MPI
  - Intel MPI 5.1
  - Intel MPI 2017

6.3.2 MPI Singularity integration

As Singularity is the final selected technology to manage containers at Finis Terrae II, its relation with High Performance Computing will be enlarged. The most critical point is the correct use of MPI.

Singularity developers ensure that their product has the ability to properly integrate with the Message Passing Interface (MPI)[7]: *"Work has already been done for out of the box compatibility with Open MPI (both in Open MPI v2.1.x as well as part of Singularity)."* Later in this document this will be tested.

Singularity makes use of a hybrid MPI container approach, this means 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 to mpirun (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. Open MPI libraries continue to open the non-ldd shared libraries (dlopen)
9. Open MPI 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 Open MPI in an HPC environment:

- Open MPI 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 Open MPI support, Open MPI version 2.1 should be used. However, Singularity team explain that there are three caveats:

1. Open MPI 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. Open MPI 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).

6.4 Container Operating System and distribution

At first, what was intended to do was to obtain different Linux distributions in each container, with different Open MPI versions in their repositories. The following commands were employed to know the available Open MPI version for each container:

Listing 1: Open MPI versions check commands

```
apt-cache policy openmpi-common
apt-get install devscripts && rmadison openmpi-common
```

The difference between `apt-cache` and `rmadison` is that `apt-cache` shows only the information known to the system (but can be used offline) while `rmadison` shows all versions of available packages from the official repositories. Thank to that, it is possible to check if some Open MPI is already installed on a container by default or not, and the Open MPI versions available on its repositories. This allow a simple installation with aptitude, whenever the package manager is APT. For example, for a Kali_2017.2-based container:

Listing 2: Open MPI versions available for Kali 2017.2

```
rmadison openmpi-common
openmpi-common | 1.4.5-1    | oldoldstable        | all
openmpi-common | 1.6.5-9.1  | oldstable-kfreebsd  | all
openmpi-common | 2.0.2-2    | stable              | all
openmpi-common | 2.1.1-6    | testing             | all
openmpi-common | 2.1.1-6    | unstable            | all
```

In this example, Open MPI versions 2.0.2 and 2.1.1 are available, so we can install them easily (e.g. `apt-get install openmpi-common/2.1.1`).

Multiple distributions were proved, like Debian, Ubuntu, Kali or Fedora. However, because the availability of the different versions of Open MPI in the official repositories was not as varied as expected, the approach was changed.

What was done in the final approach was to prepare different containers with exactly the same Operating System and installed on each one a different version of

Open MPI, but this time, downloaded from the Open MPI official page. The selected distribution was Ubuntu 16.04.2 LTS (Ubuntu Xenial) because it is the current distribution provided by the Docker Hub by default if a Ubuntu container is requested (without indicating the version). The way Open MPI is downloaded and installed was automatized in a Singularity bootstrap definition file, which can be consulted in section 14.

6.5 Employed compilers

Adding a new variable to the problem, the employed compiler in the cluster must be taken into account. The cluster has a good battery of different compilers, that is:

- GNU compilers
  - gcc/5.3.0
  - gcc/6.1.0
  - gcc/6.3.0
- Intel compilers
  - intel/2016
  - intel/2017

Changing to the compiler employed into the containers, it was that one which is included per default with the selected distribution, that is it: `5.4.0-6ubuntu1~16.04.4` (gcc/5.4.0).

6.6 PMI

The Process Management Interface (PMI)[8] has been used for quite some time as a means of exchanging wireup information needed for interprocess communication. Two versions (PMI-1 and PMI-2) have been released as part of the MPICH effort. While PMI-2 demonstrates better scaling properties than its PMI-1 predecessor, attaining rapid launch and wireup of the roughly 1M processes executing across 100k nodes expected for exascale operations remains challenging.

PMI Exascale (PMIx) represents an attempt to resolve these questions by providing an extended inter-operable version of the PMI standard specifically designed to support clusters up to and including exascale sizes. At the server side, PMIx is a part of the Open MPI run-time environment (ORTE) and the related Open Resilient Cluster Manager (ORCM). The overall objective of the project is not to branch the existing pseudo-standard definitions, in fact, PMIx fully supports both of the existing PMI-1 and PMI-2 APIs, but rather to:

- Augment and extend those APIs to eliminate some current restrictions that impact scalability.
- Provide a reference implementation of the PMI-server that demonstrates the desired level of scalability.

As the latest versions of Open MPI make use of PMIx, it is expected that different versions of MPI can be used inside and outside of the container, taking advantage of its use. E.g. probably it will be impossible to get the OMPI 1.10 series to work with anything other than itself as it pre-dates PMIx, but the OMPI 2.0 and 2.1 series should work across each other as they both include PMIx 1.x.

So, looking into the future, one thing could be done is try to build all OMPI versions against the same PMIx external library. This will ensure that the shared memory store in PMIx is compatible across the versions, and things should work since OMPI does not care how the data is moved across the host-container boundary.

7 Environment study

Now all variables have been presented, we will present the environment study in a more specific way.

Six containers were created. Five of them have Open MPI installed, while the last one was configured with Intel MPI inside. All them make use of *Ubuntu 16.04.2 LTS (Ubuntu Xenial)*, downloaded from the Docker Hub. The creation and configuration of the containers with Open MPI was completely automatized, making use of a bash script and the bootstrap definition files. By contrast, the container with Intel MPI required a much more manual installation, because of the installation process required by this software.

Remember that 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, 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 will be made 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.

Understanding a little more the number of test needed for every approach we can see the table below.

Table 1: Approximation to needed tests

Container ompi/	host ompi/				
	1.10.2	2.0.0	2.0.1	2.0.2	2.1.1
1.10.2					
2.0.0					
2.0.1					
2.0.2					
2.1.1					

As we deal with five different versions of Open MPI, their mixtures will give rise to 25 tests. But in addition, we must execute the same under different compilers, since they can be a determinant variable, and its implication will not be clear until the tests are realized. Not all host ompi configurations have the same compilers available, but they almost always have at least two different compilers. Also, we must repeat these test at least twice, one execution in just one node, and the other one in multiple nodes. This means that each approach will take approximately 100 executions. It is important to automate the process as much as possible because it is a tedious and repetitive task.

Finally, it is important to specify that on the Finis Terrae II we can use different process managers. One of them is the *srun* tool, which is part of the *Slurm*[11] resources manager, already installed on the Finis Terrae II. *Srun* is a tool employed to manage the resources and the processes. Besides, Open MPI provide *mpirun* as process manager, which offers similar functionalities than *srun*, but they are more restricted.

8 Approaches of the problem and development

8.1 Tests

The selected application for the ompi tests was `"ring"`[9]· It is one of the most basics applications, what ensures a basic distributed communication employing MPI.

It works as follow: The ring program initializes a value from process zero, and the value is passed around every single process. The program terminates when process zero receives the value from the last process. First process makes sure that it has completed its first send before it tries to receive the value from the last process. All of the other processes simply call `MPI_Recv` (receiving from their neighboring lower process) and then `MPI_Send` (sending the value to their neighboring higher process) to pass the value along the ring. `MPI_Send` and `MPI_Recv` will block until the message has been transmitted. Because of this, the `printfs` should occur by the order in which the value is passed.

For the impi tests, Intel MPI Library[10] comes with a set of source files for simple MPI programs that enable to test the installation. Test program sources are available for all supported programming languages and are located at `<installdir>/test`, where `<installdir>` is `/opt/intel/compilers_and_libraries_<version>.x.xxx/linux/mpi` by default. The installation directory for the impi containers was the default one, replacing version for 2017.4.196. The source files available into this directory are:

Listing 3: Intel MPI source files for simple MPI programs

```
.
|-- test.c
|-- test.cpp
|-- test.f
|-- test.f90
|-- Test.java
```

Since the Open MPI test program (`ring_c.c`) was written in C, the C version of the Intel test program was also chosen. To compile the selected program, we use the Intel MPI compiler, available on `/opt/intel/compilers_and_libraries_2017.4.196/linux/bin/intel64/i` The program functioning is the simplest possible: a `printf` saying Hello World in a for loop, which includes every functional unity.

8.1.1 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 and Intel MPI inside and outside the container. For this first approach, the selected process manager was *mpirun*.

8.1.1.1 MPIRUN

The first test done was the mix of the different ompi possibilities, including the different available compilers. For these executions, the obtained results were a one-to-one version compatibility. That is, for the properly program execution, the version of Open MPI must match exactly inside and outside the container. However, the same conclusion was obtained for the different conjugations with the different compilers available. This means that, until proven otherwise, the compiler will not be in itself a limiting factor. In addition, for this particular environment, the results were the same for single node cases as for multiple node cases. Results can be visualized on table 2.

Table 2: MPIRUN mixing different compilers, 1 and 2 nodes. Open MPI tests.

Container ompi/	host ompi/				
	1.10.2	2.0.0	2.0.1	2.0.2	2.1.1
1.10.2	✓	✗	✗	✗	✗
2.0.0	✗	✓	✗	✗	✗
2.0.1	✗	✗	✓	✗	✗
2.0.2	✗	✗	✗	✓	✗
2.1.1	✗	✗	✗	✗	✓

The same configuration was employed to prove the impi possible configurations. In this case, the selected program was `test.c` instead of `ring_c.c`. Due to Intel MPI 5.1 version is no longer available to download on the Intel Developer Zone, only a container impi/2017 could be prepared. In order to test an impi/5.1 container, we use a mirror of a compute node with impi/5.1 on it. Althout this is not exactly the same as prepare a brand new container and install on it the Intel MPI library, it is very close to it. The obtained results were the same than witch the ompi tests, only one-to-one matches worked properly. This can be observed on table 3.

Table 3: MPIRUN mixing different compilers, 1 and 2 nodes. Intel MPI tests.

Container impi/	host impi/	
	5.1	2017
5.1	✓	✗
2017	✗	✓

Note: In order to set some boundaries to the study, from this moment we decide to study only the Open MPI compability, avoiding impi distributions.

8.1.1.2 SRUN

Continuing the first presented approach, now we will execute the exactly same tests, but under the *srun* process manager.

As could be expected, results were not the same. The big difference is that under the *srun* process manager, the multiple nodes executions were not successful. Only executions under one node and with one-to-one matches for the Open MPI versions inside and outside the container were successful. This behavior can be observed on table 4.

Table 4: SRUN mixing different compilers, 1 node. Open MPI tests.

Container ompi/	host ompi/				
	1.10.2	2.0.0	2.0.1	2.0.2	2.1.1
1.10.2	✓	✗	✗	✗	✗
2.0.0	✗	✓	✗	✗	✗
2.0.1	✗	✗	✓	✗	✗
2.0.2	✗	✗	✗	✓	✗
2.1.1	✗	✗	✗	✗	✓

The results for multiple nodes can be observed on table 5. When this test was done, false positives were obtained for every one-to-one match. In those cases, when the ring program was executed, it did an execution for every node, but none of them could "see" the others. Because of this reason, the ring call was executed as many instances as nodes were indicated, having each of them just one process per ring. So, parallel communication was not obtained.

Table 5: SRUN mixing different compilers, 2 nodes. Open MPI tests.

Container ompi/	host ompi/				
	1.10.2	2.0.0	2.0.1	2.0.2	2.1.1
1.10.2	✗	✗	✗	✗	✗
2.0.0	✗	✗	✗	✗	✗
2.0.1	✗	✗	✗	✗	✗
2.0.2	✗	✗	✗	✗	✗
2.1.1	✗	✗	✗	✗	✗

Remember there were some important considerations to work with MPI:

- Open MPI 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.

We have made sure to meet all these requirements to obtain the correct operation of MPI, as indicated by the Singularity team. All containers support InfiniBand and PMI, thanks to the OFED libraries installed during the container creation process. Theoretically, under this environment, all tests made under ompi containers with a lower Open MPI version than the cluster Open MPI version employed in each moment should work properly. However, attending to the results, we can ensure that after a series of practical tests, crossversion Open MPI compatibility can not be obtained making simple mixes. After the realization of these tests, a dependency with the PMI library was detected. We also need to use exactly the same PMI version linked with the Open MPI in both, the host and the container.

After this, the simple mixing approach was abandoned, and we turned into a new approach.

8.1.2 Binding libraries and their dependencies

Since the previous approach, which consists of simply mixing the different MPI versions did not work as expected, we will give it a twist and we will apply a completely different approach. Singularity containers allow us to bind some files and folders as we want to from the host. So, we will try to prepend the Open MPI libraries inside the container, as well as all their dependencies, with the Open MPI libraries installed in the host. Then, what we are trying to do at all is to employ exactly the same version of Open MPI inside and outside the container, what we already know it works properly. With this approach we force the compatibility of the entire Open MPI environment, Open MPI itself and its dependencies.

Its advantage is that it will be absolutely transparent for the final user. When a container arrive, it will be executed into a environment which will bind its Open MPI libraries to make use of the cluster Open MPI libraries, instead of their own, right inside the container. In counterpart, the big and obvious disadvantage is that the container philosophy will be corrupted. Containers will pass from use their own resources to use those what belong to the host, what supposes the custom adaptation of the container in each host to get the desired portability. With this I mean that if we want to use this approach on a different cluster, we must adjust the libraries binding for every unique case. However, as we execute this solution under the Finis Terrae II, it could suppose a solution that contributes with a high level of flexibility and portability. That is, the cluster should accept a big and varied quantity of containers which make use of Open MPI. Likewise, it is very important to realize that the containers alone will not represent any loss of portability under any concept, because the customization of the containers consists only in a hierarchy of directories where to bind the libraries. That is, the same containers that could be employed in the Finis Terrae II under this solution are perfectly usable in other clusters or machines, without requiring any direct change in them.

Browsing the Singularity web we can found suggestion of how the binding should be done[12]. Although this recommendations are made for GPU drivers, Open MPI binding is named too as an alternative configuration solution. The explication is not very exhaustive, but at least it is considered by the Singularity team. Several test must be done in order to achieve the final binding solution.

8.1.2.1 SRUN

This approach was made under the *srun* process manager. A exhaustive study has been done to detect all the Open MPI dependencies, all of them located 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.

Listing 4: Binding folders hierarchy

```
host
|-- lib64
|-- mpi
|   `-- lib
|       |-- ompi_1.X
|       `-- ompi_2.X
`-- usr
    `-- lib64
```

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). If the reader wishes, he can track this compatibility loss making use of the ABI tracker tool[13].

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 3.

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.

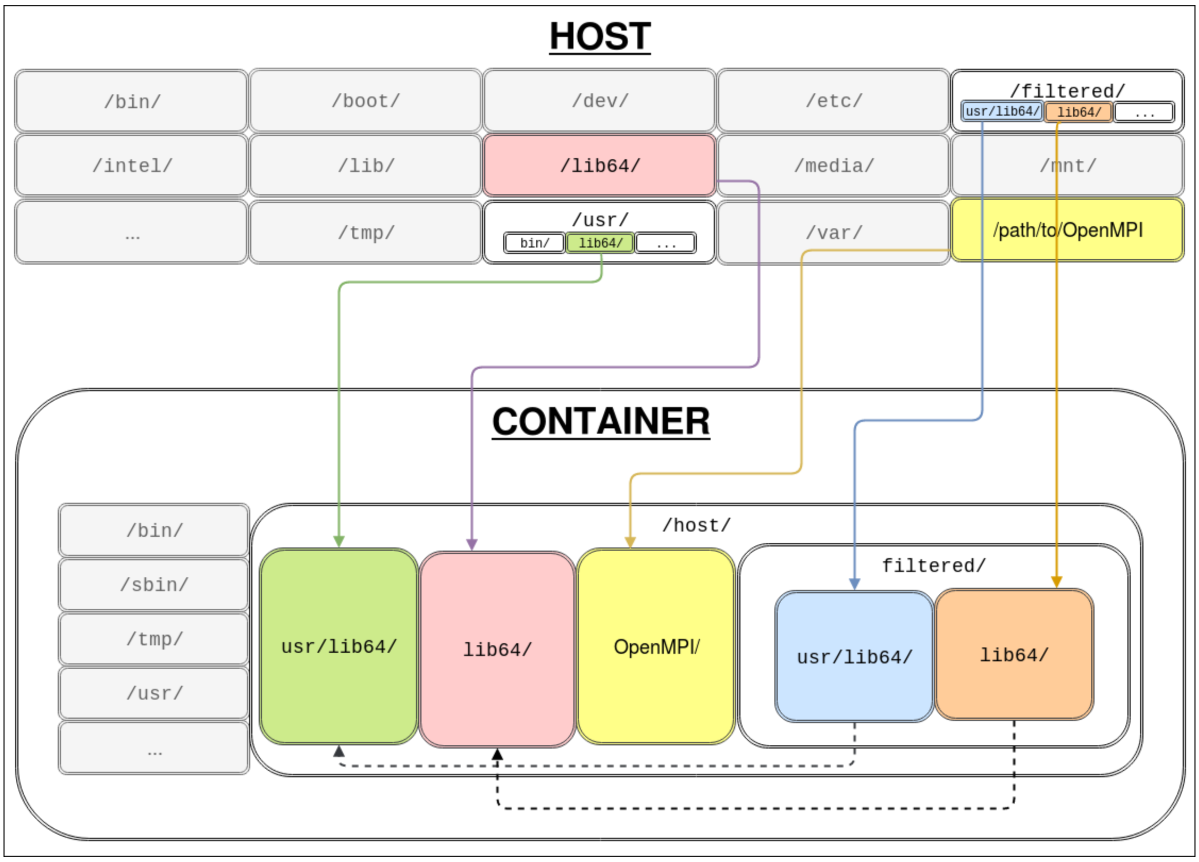

Figure 3: Binding approach: graphic representation

Explaining the figure 3 in a more structured way, this is what we do:

1. Bind the Open MPI 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` prepending the binded Open MPI libraries, dependencies and directories with symbolic links.

9 Binding approach: How to

To understand better what was done in this approach, I will show a little example. This concrete example was done under a container ompi/2.0.1. First of all, we call `ldd` inside the container to find important Open MPI dependencies:

Listing 5: ldd example

```
> ldd /usr/bin/ring
        linux-vdso.so.1 =>  (0x00007fff3edb4000)
        libmpi.so.20 => /usr/lib/libmpi.so.20 (0
           ↪ x00007f6407009000)
        libpthread.so.0 => /lib/x86_64-linux-gnu/
           ↪ libpthread.so.0 (0x00007f6406de6000)
        libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6
           ↪ (0x00007f6406a1b000)
        libopen-rte.so.20 => /usr/lib/libopen-rte.so
           ↪ .20 (0x00007f6406700000)
        libopen-pal.so.20 => /usr/lib/libopen-pal.so
           ↪ .20 (0x00007f64062d4000)
        librt.so.1 => /lib/x86_64-linux-gnu/librt.so.1
           ↪  (0x00007f64060cb000)
        libm.so.6 => /lib/x86_64-linux-gnu/libm.so.6
           ↪ (0x00007f6405dc2000)
        /lib64/ld-linux-x86-64.so.2 (0
           ↪ x00007f640741f000)
        libibverbs.so.1 => /usr/lib/libibverbs.so.1 (0
           ↪ x00007f6405bb3000)
        librdmacm.so.1 => /usr/lib/x86_64-linux-gnu/
           ↪ librdmacm.so.1 (0x00007f640599b000)
        libnuma.so.1 => /usr/lib/x86_64-linux-gnu/
           ↪ libnuma.so.1 (0x00007f6405790000)
        libpmi.so.0 => /usr/lib/x86_64-linux-gnu/
           ↪ libpmi.so.0 (0x00007f640558a000)
        libutil.so.1 => /lib/x86_64-linux-gnu/libutil.
           ↪ so.1 (0x00007f6405386000)
        libdl.so.2 => /lib/x86_64-linux-gnu/libdl.so.2
           ↪  (0x00007f6405182000)
        libslurm.so.29 => /usr/lib/x86_64-linux-gnu/
           ↪ libslurm.so.29 (0x00007f6404e1c000)
```

Observe to the dependency: `libnuma.so.1`. This is a known MPI library dependency, so it must be replaced by the host one. Then, we search its host equivalent:

Listing 6: Search host library equivalent

```
> find / -iname libnuma.*
        /usr/lib64/libnuma.so
        /usr/lib64/libnuma.a
        /usr/lib64/libnuma.so.1
```

As we found its host equivalent, we must move (or copy) it to the binding path.

Now the symbolic link is inside the folder which will be binded, the container will use it instead of its own one. As we cover dependencies, we will discover new ones. We must repeat the process as many times as necessary. For this, we can use `ls` with some options. For example, for the present case:

Listing 7: Discovering new dependencies

```
> ls -lia /host/usr/lib64/libnuma.so.1
```

In addition, it is sometimes necessary to create links between files, because the library is not directly linking to the version of the library that we need. For example, for the `libmpi` library there are numerous versions that we would already have prepared thanks to binding.

Listing 8: Multiple library versions

```
> find . -iname libmpi.*
    ./host/mpi/lib/ompi_1.X/libmpi.so
    ./host/mpi/lib/ompi_1.X/libmpi.so.20
    ./host/mpi/lib/ompi_1.X/libmpi.so.12
    ./host/mpi/lib/ompi_1.X/libmpi.la
    ./host/mpi/lib/ompi_1.X/libmpi.so.12.0.2
    ./host/mpi/lib/ompi_2.X/libmpi.so
    ./host/mpi/lib/ompi_2.X/libmpi.so.20.0.1
    ./host/mpi/lib/ompi_2.X/libmpi.so.20
    ./host/mpi/lib/ompi_2.X/libmpi.so.12
    ./host/mpi/lib/ompi_2.X/libmpi.la
```

After a trial error test, we find that, for example, this library by default is not linking to what we want, so we can force this link thanks to `ln`.

Listing 9: Creating links between files

```
> ln -s libmpi.so libmpi.so.20
```

It is necessary to repeat these steps until no more dependencies are shown. When this moment comes, the application should work properly. Once all dependencies were convered, the execution tests were repeated. Results can be shown at the table below.

Table 6: srun ring 2 nodes, mixing compilers

Container ompi/	host ompi/				
	1.10.2	2.0.0	2.0.1	2.0.2	2.1.1
1.10.2	✓	⚠	⚠	⚠	⚠
2.0.0	✓	✓	✓	✓	✓
2.0.1	✓	✓	✓	✓	✓
2.0.2	✓	✓	✓	✓	✓
2.1.1	✓	✓	✓	✓	✓

As can be seen at the table above, once all the involved libraries and its dependencies were satisfied, the test program, ring, could be executed under any container-FT2 Open MPI mix. However, if container ompi/1.10.2 is used with a higher Open MPI version outside, it will show a warning prompt, making reference to different sizes in symbols (e.g. Symbol 'ompi_mpi_comm_world' has different size in shared object, consider re-linking). Due to this, future real world applications could not work properly or crash.

In addition, we must keep in mind that other applications can make use of some libraries that the test program has not done. In that case, we should repeat all the above steps to cover all dependencies again.

Leaving aside these small possible incidents, it seems that we have come up with a solution that allows the use of different versions of Open MPI in and out of containers. This means that when any container with an application that uses MPI arrives, the cluster should be able to run it without problem, whenever it does under this environment.

All the scripts employed to achieve this final solution, as well as the filtered libraries hierarchy can be consulted on section 14.

10 Real world applications

Once the test applications passed; real world applications, built inside already created containers, were proved. For this reason, the Open MPI version contained could not be changed. As we achieved a solution what seems work properly, we will make use of it for these executions. That is, we will check if our solution is suitable for big and complex problems. As these containerized programs belong to real world solutions, their proper execution should suppose a quality assurement for our achieved approach. We will test programs written in the following programing languages: *C++*, *Fortran* and *Python*. All them were executed under two distributed nodes.

10.1 C++ application: Feel++

Table 7: Feel++

Container ompi/	host ompi/				
	1.10.2	2.0.0	2.0.1	2.0.2	2.1.1
1.10.2	✓	✓	✓	✓	✗

As can be seen in this test, a container ompi/1.10.2 worked over 1.10.2, 2.0.0, 2.0.1 and 2.0.2 outside Open MPI versions. Nevertheless, host ompi/2.1.1 did not work. Most robably this is because the latest version employs an Intel compiler, and not a GNU compiler as the other ones. So, for the very first time, **a compiler dependency was discovered**.

10.2 Fortran application: XH5For

Table 8: XH5For

Container ompi/	host ompi/				
	1.10.2	2.0.0	2.0.1	2.0.2	2.1.1
1.10.2	✓	✓	✓	✓	✗
2.1.1	✓	✓	✓	✓	✗

Confirming the previous named dependency, the Fortran application worked over all host ompi except ompi/2.1.1, which makes use of an Intel compiler. This bahavior is observed for different container ompi versions, even when there is a one-to-one match, because the containerized application was compiled with a GNU compiler.

10.3 Python application: Distributed Smith-Waterman

To get the container running applications that use MPI with Python, it is necessary to install a new library, MPI4Py[14], which has followed the next process of installation and configuration inside the container.

Listing 10: Installing MPI4Py

```
cd /tmp/
git clone https://github.com/mpi4py/mpi4py.git
cd mpi4py/
# Superuser permissions needed
python setup.py build
python setup.py install
cp -rf /tmp/mpi4py/ /usr/bin/
```

Since Open MPI was installed in the default directory, MPI4Py will automatically locate it and perform the configuration itself. If you have installed Open MPI in a different directory, you would need to explicitly specify the installation path.

Finishing the real world applications tests, a Python application was executed. It is an implementation of the Smith-Waterman algorithm[15], adaptated to work in a distributed parallel mode. Its functioning is not very complicated, it is formed by send, receive and barrier operations. If the reader wishes, he can deepen the operation of the algorithm by consulting the employed source file[16].

Table 9: Smith-Waterman

Container ompi/	host ompi/				
	1.10.2	2.0.0	2.0.1	2.0.2	2.1.1
1.10.2	✓	✓	✓	✓	✓
2.1.1	✓	✓	✓	✓	✓

Unlike the other actual tested applications, in this case the operation was correct under all host ompi combinations. It is also true that the implementation of this application is simpler compared to the previously presented, but we must keep in mind that Python is usually used for the creation of simple programs on HPC.

11 Benchmarks

One of the biggest containers' advantages is its theoretical efficient and lightweight operating mode. When using the same kernel as the host machine, and avoiding replication as much as possible, makes of containers virtualization a great solution for HPC programs execution.

Nevertheless, until this lightness is verified in our work environment, we will not stop talking about a completely theoretical framework. Because of this, some benchmarks will be done, to verify this in a practical way.

So, the objective of the following benchmark tests is no other than proof that containers make a propper use of HPC resources like memory, networking or computing.

11.1 InfiniBand network benchmarks: OSU microbenchmarks

Since the goal of this document is the description of how to use Singularity containers employing different versions of MPI, one of the most interesting benchmarks that can be done is the use of the network. We are not talking about a network that connects with the outside, but an internode network that allows the communication of the different processors for a correct distribution operation. For this purpose, Finis Terrae II, as well as many others clusters, uses an InfiniBand network. InfiniBand is a computer-networking communications standard used in high-performance computing that features very high throughput and very low latency. It is used for data interconnect both among and within computers.

So, now the Singularity containers are using MPI, we will check if they are making use of the InfiniBand network for that.

The selected bechmarks for this practical approach are the *OSU microbenchmarks*[17], belonging to *The Ohio State University*, what will allow us to measure some interesting parameters, such as latency or bandwidth. Once the tests were done, we will check if the obtained results belong to a reasonable InfiniBand network parameters. The objective of this test is not to obtain the best possible results in relation to the network, but only to see if the containers are actually making use of the InfiniBand network, so no real optimizations were done with the network. As we do not care about obtainning the best values, the results of these tests will not be compared with any others, like the values under a native execution, we will only check their belonging to reasonable InfiniBand values.

When the *OSU microbenchmark* pack is downloaded in our container, it can be compilled and installed easily thanks to the included makefile.

Listing 11: Installing OSU microbenchmarks

```
sudo singularity exec -w CONTAINER_NAME.img ./
   ↪ configure CC=/path/to/special/mpicc --prefix=<
   ↪ path-to-install> && make && make install
```

Once the *OSU microbenchmarks* are installed, we can find a battery of test in the directory `/usr/bin/libexec/osu-micro-benchmarks` (default installation directory). This battery of benchmarks is composed by the following:

Listing 12: OSU microbenchmarks

```
.
`-- osu-micro-benchmarks
    `-- mpi
        |-- collective
        |   |-- osu_allgather
        |   |-- osu_allgatherv
        |   |-- osu_allreduce
        |   |-- osu_alltoall
        |   |-- osu_alltoallv
        |   |-- osu_barrier
        |    |-- osu_bcast
        |    |-- osu_gather
        |    |-- osu_gatherv
        |    |-- osu_iallgather
        |    |-- osu_iallgatherv
        |    |-- osu_ialltoall
        |    |-- osu_ialltoallv
        |    |-- osu_ialltoallw
        |    |-- osu_ibarrier
        |    |-- osu_ibcast
        |    |-- osu_igather
        |    |-- osu_igatherv
        |    |-- osu_iscatter
        |    |-- osu_iscatterv
        |    |-- osu_reduce
        |    |-- osu_reduce_scatter
        |    |-- osu_scatter
        |    `-- osu_scatterv
        |-- one-sided
        |    |-- osu_acc_latency
        |    |-- osu_cas_latency
        |    |-- osu_fop_latency
        |    |-- osu_get_acc_latency
        |    |-- osu_get_bw
        |    |-- osu_get_latency
        |    |-- osu_put_bibw
        |    |-- osu_put_bw
        |    `-- osu_put_latency
        |-- pt2pt
        |    |-- osu_bibw
        |    |-- osu_bw
        |    |-- osu_latency
        |    |-- osu_latency_mt
        |    |-- osu_mbw_mr
        |    `-- osu_multi_lat
        `-- startup
            |-- osu_hello
            `-- osu_init
```

The working of these different benchmarks can be consulted on the README file which is downloaded along with the benchmarks. If we do not want to download the *OSU microbenchmarks* package to read this file, it can be also consulted online[18].

As can be seen, a lot of different benchmarks are available on this package. We will focus our attention on the point-to-point (pt2pt) tests, what will allow us to measure and to evaluate the network's characteristics from the own containers.

Having understood the purpose of the tests to be performed, we proceed to explain their specific behavior as well as how they will be executed. We will make use of a little battery of five containers with Ubuntu 16.04.2 LTS (Ubuntu Xenial). Each container will have installed inside different Open MPI versions: 1.10.2, 2.0.0. 2.0.1, 2.0.2 and 2.1.1. Then, the *OSU microbenchmarks* will be executed under the Finis Terrae II employing these containers, and mixing them with different Open MPI versions on it (outside the containers). The selected outside Open MPI versions are: 1.10.2, 2.0.0 and 2.0.1. So, a total of fifteen executions per test will be done, in order to obtain more stable values.

To ensure the use of the InfiniBand network, the obtained values will be compared with some scientific papers that tell how they used the same tests to measure their own InfiniBand network. These texts belong to research centers or similar, so they will be considered completely reliable sources. Some of them compare the InfiniBand network values versus the Ethernet network values obtained under the same tests. So, if the results obtained under the Finis Terrae II hover around similar values, the InfiniBand network use will be assured.

Once the different benchmarks were executed, it is easily observable that the obtained results under the same Open MPI version out of the container are very stable for the five Open MPI version combinations within the container. What has been done is to perform the average of the values belonging to the same version of Open MPI outside the container, to perform a comparison between the three versions of Open MPI outside the container.

11.1.1 Latency Tests

The theoretical latency values of InfiniBand systems is about 160 nanoseconds. The real ones are around 6 microseconds, depending a lot on the software and firmware. So, we expect to obtain latency values around 6 microseconds.

11.1.1.1 osu_latency - Latency Test The latency tests are carried out in a ping-pong fashion. The sender sends a message with a certain data size to the receiver and waits for a reply from the receiver. The receiver receives the message from the sender and sends back a reply with the same data size. Many iterations of this ping-pong test are carried out and average one-way latency numbers are obtained. Blocking version of MPI functions (MPI_Send and MPI_Recv) are used in the tests.

As can be shown in figure 4, all tests showed very stable results regardless of the version of Open MPI used. The latency values shown are very small, less than 6 microseconds with window sizes up to 8192 bytes.

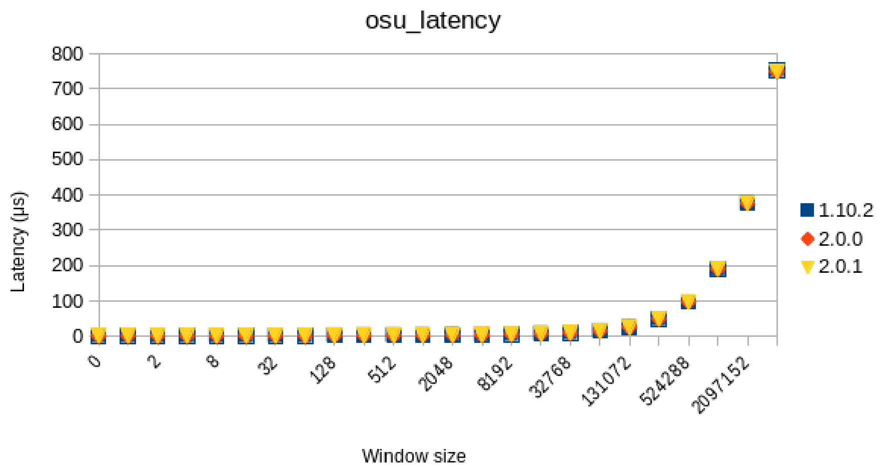

Figure 4: OSU Latency Test

11.1.1.2 osu_multi_lat - Multi-pair Latency Test This test is very similar to the latency test. However, at the same instant multiple pairs are performing the same test simultaneously. In order to perform the test across just two nodes the host-names must be specified in block fashion.

For the execution of this test, four cores were employed, in four different nodes. Results can be observed in figure 5. It seems that by extending the complexity of the problem, making multiple pairs to perform the same test simultaneously, at the same instant, does not increase latency. The obtained results show one more time an extraordinary stability and a very low latency.

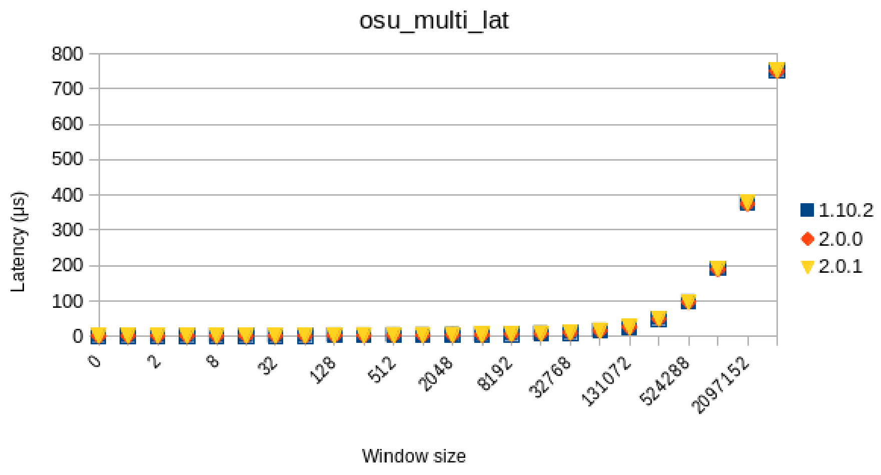

Figure 5: OSU Multi-pair Latency Test

11.1.2 Bandwidth Tests

11.1.2.1 osu_bw - Bandwidth Test The bandwidth tests were carried out by having the sender sending out a fixed number (equal to the window size) of back-to-back messages to the receiver and then waiting for a reply from the receiver. The receiver sends the reply only after receiving all these messages. This process is repeated for several iterations and the bandwidth is calculated based on the elapsed time (from the time sender sends the first message until the time it receives the reply back from the receiver) and the number of bytes sent by the sender. The objective of this bandwidth test is to determine the maximum sustained date rate that can be achieved at the network level. Thus, non-blocking version of MPI functions (MPI_Isend and MPI_Irecv) were used in the test.

The general behavior of this test is the expected and desired. It can be observed in figure 6 As the window size increases, the bandwidth required for transmission is increased, until the point at which the maximum bandwidth is reached, which is near 6000 MB/s. In general, the different versions of Open MPI outside the container do not represent a differentiating factor in terms of bandwidth. However, in the central values it is possible to observe a slight advantage under the use of Open MPI 1.10.2. Anyway, in the "critical" values, which could be considered those in which the bandwidth is being consumed in its entirety, they all have the same behavior.

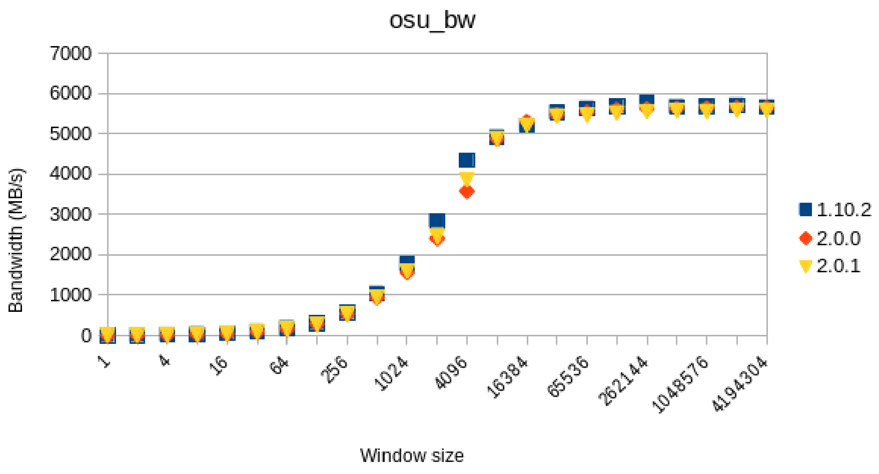

Figure 6: OSU Bandwidth Test

11.1.2.2 osu_mbw_mr - Multiple Bandwidth The multi-pair bandwidth and message rate test evaluates the aggregate uni-directional bandwidth and message rate between multiple pairs of processes. Each of the sending processes sends a fixed number of messages (the window size) back-to-back to the paired receiving process before waiting for a reply from the receiver. This process is repeated for several iterations. The objective of this benchmark is to determine the achieved bandwidth and message rate from one node to another node with a configurable number of processes running on each node.

Now the number of cores involved in the problem will be increased. A total of four cores (belonging to four different nodes) were used. So, the waited behavior is to have two senders and two receivers, what will cause the obtaining of double the bandwidth, compared to the uni-directional test, since the result are added. Fortunately this is what it was obtained as result, as can be seen in figure 7.

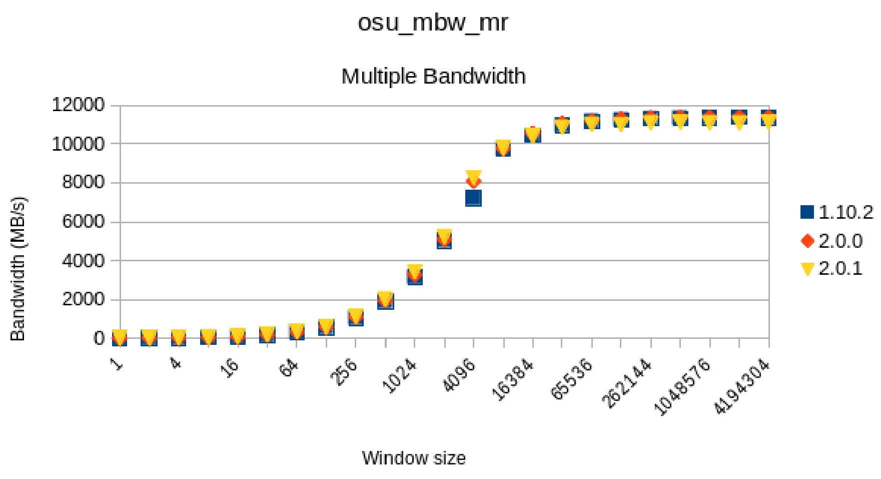

Figure 7: OSU Multiple Bandwidth Test

11.1.2.3 osu_mbw_mr - Message Rate Test Analyzing the message rate, if we deal with small sized messages, such as those shown at the beginning of the graph 8, a big number of them can be sent, thanks to the big bandwidth available. As its size increases, the bandwidth will not be enough to send so many. Therefore, the behavior should be approximately the reverse of the 7 graph (understanding behavior as figure shape, not its values!). As an add-on, it can be said that for smaller sized messages as the outside Open MPI version is higher, its behavior is getting better.

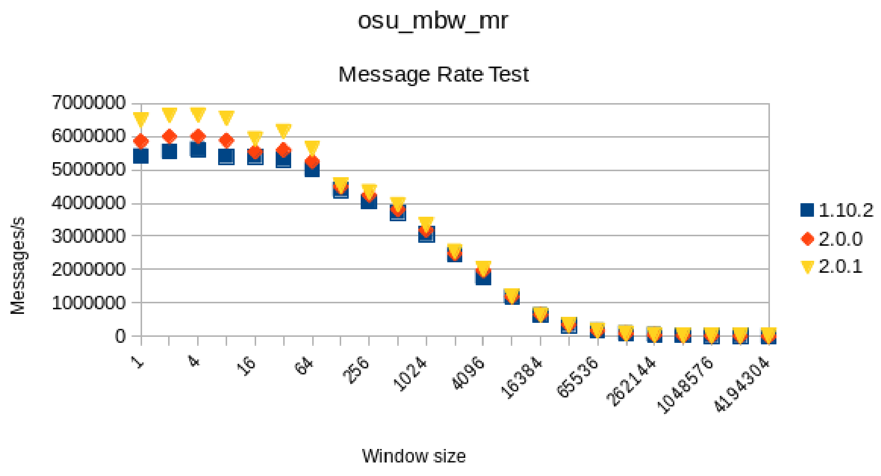

Figure 8: OSU Message Rate Test

Now all the result test are available, it is clear that the InfiniBand network is been used. Parameters as the very low latency (6 microseconds or less) are critical to reach this conclusion. Furthermore, the obtained results are in agreement with those obtained by other teams using the same tests, like *UL HPC MPI Tutorial: Building and Runnning OSU Micro-Benchmarks* by the *University of Luxembourg High Performance Computing (HPC) Team*[19]; *Some Micro-benchmarks for HPC: 10Gb Ethernet on EC2 vs QDR InfiniBand*[20], by Adam DeConinck, HPC Cluster Administrator at *Los Alamos National Laboratory*; or *Implementation and comparison of RDMA over Ethernet*[21], belonging to *National Security Education Center, Los Alamos National Laboratory*. Reader is free to consult such documents if he wants further information. **It seems that containers with different inside-outside Open MPI versions make use of the InfiniBand network.**

11.2 RAM benchmarks: STREAM

The *STREAM* benchmark[22] is a simple synthetic benchmark program that measures sustainable memory bandwidth (in MB/s) and the corresponding computation rate for simple vector kernels. The reason why the *STREAM* benchmark was the selected, and not another one, is because *STREAM* is the de facto industry standard benchmark for measuring sustained memory bandwidth. It uses data sets much larger than the available cache on a system, which avoids large amounts of time devoted waiting for cache misses to be satisfied. Because of this reason, *STREAM* documentation indicates that the value of the `STREAM_ARRAY_SIZE` must be at least four times larger than the combined size of all last level caches used in the run. But this is not the only limitation to define the value of `STREAM_ARRAY_SIZE`: Its size should be large enough so that the "timing calibration" output by the program is at least 20 clock-ticks. Example: most versions of Windows have a 10 millisecond timer granularity. 20 "ticks" at 10 ms/tic is 200 milliseconds. If the chip is capable of 10 GB/s, it moves 2 GB in 200 msec. This means the each array must be at least 1 GB, or 128M elements.

To run the Stream Benchmark on multiprocessor, there are several choices: OpenMP, pthreads, and MPI. Pursuing a global consistency, we will use the MPI version of *STREAM*[1], instead of its default version, which uses OpenMP. Thank of this version, we will be able to measure the employed RAM under a multiprocessor multinode container environment, using an InfiniBand network, that is, after all, the goal of this research. This will require MPI support installed, an already well-known requirement.

If the reader of this document wishes to go further into the operation of STREAM, online official documentation[23] can be consulted.
Turning to our particular case, as could be shown in section 3, Finis Terrae II nodes have 24 cores, with the following memory cache structure:

- L1d cache: 32K
- L1i cache: 32K
- L2 cache: 256K
- L3 cache: 30720K

So, attending to the previous limitations to define the value of the `STREAM_ARRAY_SIZE`, it is important to consider the values of the last level cache, as well as the number of available cores. As a plus requirement, added because of the MPI use, two different nodes will be used, in order to ensure the MPI and InfiniBand network utilization. Besides, it is important to reserve the nodes by they full (employing the 24 cores per node and all the available RAM), obtaining the benchmark execution in a blocking way under this environment, without other parallel processes that could generate noise. Doing some arithmetic, we can observe that the sum total of the last level cache per node is 720MiB (30720K = 30MiB per core, having 24 cores). Then, a 720MiB four times size will be needed (2880MiB should be a correct array size), considering a simple-node execution. Since we will execute the benchmark using two nodes, we will have twice the cache: 1440MiB on its last level, obtaining now a size of 5760MB as a great value for the `STREAM_ARRAY_SIZE`. So, I will define the value of the `STREAM_ARRAY_SIZE` as 760000000, which should be a value larger enough for our purposes.

The other limitation factor, the "timing calibration" output by the program, can not be considered until the program is executed, so it will be ignored at the moment.

Continuing with the values under the benchmark will be done, it is important to consider that STREAM runs each kernel `NTIMES` times and reports the best result for any iteration after the first. The selected value for this variable will be the default one: 10 times.

Finally, keep in mind that we must compile the code with optimization and enabling the correct options to be able to use multiprocessing; employing Open MPI mpicc in our case. Array size can be set at compile time without modifying the source code for the (many) compilers that support preprocessor definitions on the compile line.

[1]Specifically, stream_mpi.c, the version of STREAM MPI coded in C, without more reason than the knowledge of the language by the writter of this document.

Listing 13: Compilation example employing Singularity containers

```
sudo singularity exec -w CONTAINER_NAME.img mpicc -m64
    ↪  -o /usr/bin/stream_mpi -O -DSTREAM_ARRAY_SIZE
    ↪ =760000000 stream_mpi.c
```

Once the test were executed and the results were obtained, we can analyze them and obtain a series of conclusions. First of all, we will check the values that has to do with the array size and its implications.

- Total Aggregate Array size = 760000000 (elements)
- Total Aggregate Memory per array = 5798.3 MiB (= 5.7 GiB).
- Total Aggregate memory required = 17395.0 MiB (= 17.0 GiB).
- Data is distributed across 48 MPI ranks
  - Array size per MPI rank = 15833333 (elements)
  - Memory per array per MPI rank = 120.8 MiB (= 0.1 GiB).
  - Total memory per MPI rank = 362.4 MiB (= 0.4 GiB).

As can be seen, the execution worked over a 760000000 elements array and over 48 different cores, employing MPI. This ensures the wished multinode multicore execution. Due to these parameters, some concrete size values are obtained; e.g. a total aggregate memory value of 17GiB.

The second important execution output which must be reasoned is the timer granularity, inasmuch as it was previously defined as a limitation factor. The output value for the granularity/precision appears to be 1 microseconds, what will make each test to take on the order of 54269 microseconds (= 54269 timer ticks). Turning back our attention to the needed premises for a great execution, we can check that the array size should be large enough so that the "timing calibration" output by the program is at least 20 clock-ticks. As 54269 » 20, we can conclude the array size is large enough, and the results will suppose a good study sample.

The obtained results for the defined test are:

Table 10: Container STREAM benchmark results

Function	Best rate (MB/s)	Avg time	Min time	Max time
Copy	155550.7	0.078218	0.078174	0.078288
Scale	154837.6	0.078575	0.078534	0.078641
Add	177616.6	0.102739	0.102693	0.102845
Triad	177573.8	0.102803	0.102718	0.103148

Continuing the memory benchmark study, a native execution will be done[2], in order to compare both executions, the containerized one and the native one. Obviously, the parameters under the native execution will be done are exactly the same than those which were used in the previous containerized execution. That is why they will not be analyzed again. The obtained results under the native execution are:

[2]In this case, the `stream_mpi.c` program were compiled using the GNU compiler gcc/5.3.0, the 1.10.2 version of Open MPI, and enabling the optimization compiling flags.

Table 11: Native STREAM benchmark results

Function	Best rate (MB/s)	Avg time	Min time	Max time
Copy	155478.6	0.078266	0.078210	0.078386
Scale	154717.4	0.078623	0.078595	0.078691
Add	177729.3	0.102662	0.102628	0.102683
Triad	177710.3	0.102694	0.102639	0.102851

Note that the test results have an avg error less than 1.000000e-13 on all three arrays for both cases, what make them a reliable source. This value is given by the STREAM developer team.

Now that all the results were obtained, we can easily compare them. Starting with those related to the execution times, observe that they present values with differences that hover around the magnitude of 1.0e-4 seconds. In addition, there is no execution that always obtains the best results (less time), but in some cases this value is obtained by the native execution and in others by the containerized solution. Looking now at the bandwidth rate, there is no a clear difference between the results to assign a solution better than the other one.

So, after all the done tests and the obtained results, we can say that a containerized solution will make use of the machine memory that will come very close to the use that would be made under a native execution. **Native and container executions are equivalent in terms of memory.**

11.3 CPU benchmarks: HPL

HPL[24], a portable implementation of the *High-Performance Linpack Benchmark* for distributed-memory computers, is a software package that solves a random dense linear system in double precision arithmetic on distributed-memory computers.

The *HPL* package provides a testing and timing program to quantify the accuracy of the obtained solution as well as the time it took to compute it. It requires the availibility on the system of an implementation of the Message Passing Interface (MPI). An implementation of either the Basic Linear Algebra Subprograms BLAS or the Vector Signal Image Processing Library VSIPL is also needed. These requisites make *HPL* an excelent CPU usage measurer candidate for our concrete research.

The end of this benchmark it is not to reach the infrastructure peak of performance, but what is pursued is to verify the functioning and to check the correct performance mixing different MPI versions inside-outside containers to make a comparison with an one-to-one inside-outside MPI versions execution, already done.

What will be done is to reproduce a series of executions that were already made in the cluster, under an one-to-one inside-outside MPI versions. Thus, once all the results are obtained, both behaviors will be compared, and it will be reasoned if the solution with mixing MPI versions containers supposes, if it exists, a reasonable delay.

In order to get a battery of results a little varied, what will be done is to execute these benchmarks with an upward focus. By this I mean that I will reproduce its execution making use of a greater number of processors each time. Computations will be run in 1, 2, 4, 8, 16 and 32 nodes at Finis Terrae II. For every node increment, the problem's size will increase too, in order to take always approximately 80-90% of available memory of the involved nodes. This is what is called a scalability test weak.

The results of the one-to-one inside-outside MPI versions weak scaling tests show an increase of the aggregated performance as we increase the number of nodes involved. Looking at the results we can also see that the performance per node is maintained almost immutable along the different executions. These results are also very close to the expected theoretical values.

Figure 9 shows the performance (logarithmic scale) of the weak scaling test depending on the number of nodes involved in the computation.

To provide another view of the obtained values, they can also be seen in figure 10, with a different scale for the vertical axis (GFlops).

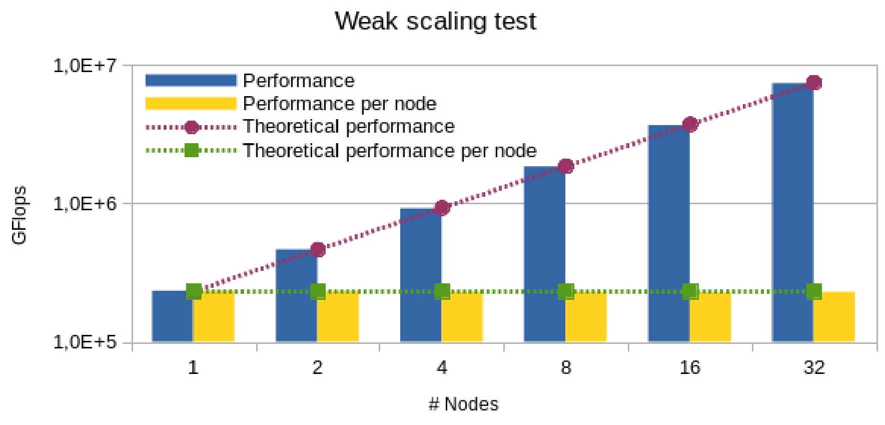

Figure 9: Weak scaling test 1

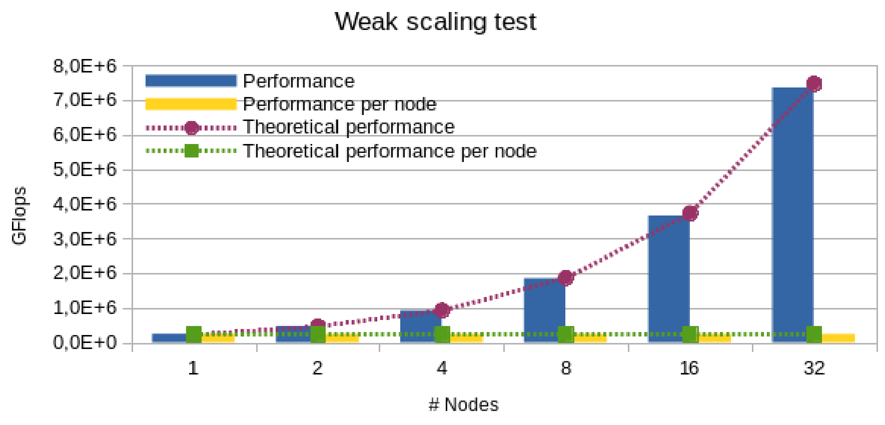

Figure 10: Weak scaling test 2

Comparing those results with the inside-outside mixed MPI execution, it is clear that both have the same trend. However, in the mix test, a little delay is present.

At this point, it is important to clarify that the test has been performed only once (a single execution), due to the large computational time that is required for this. Therefore, in the absence of comparisons with other executions, it is possible that this particular case has produced factors that generate noise, thus causing this delay. So, if we look at figures 11 and 12, we will be able to appreciate this little delay, especially on the 32 nodes execution.

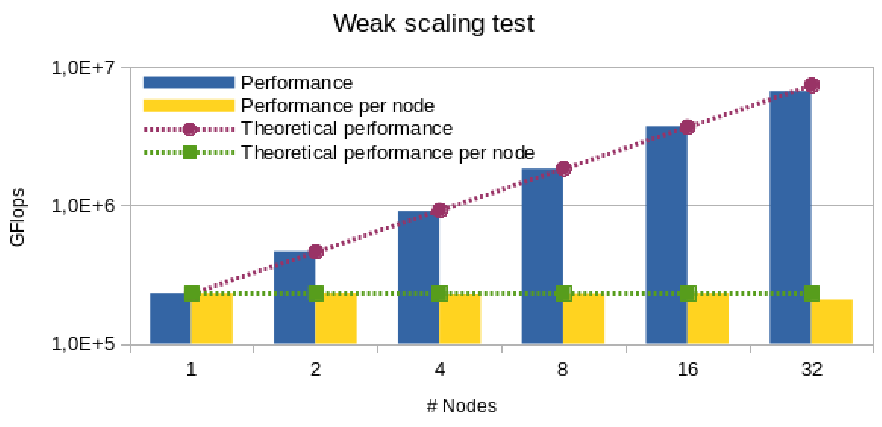

Figure 11: Weak scaling test 3

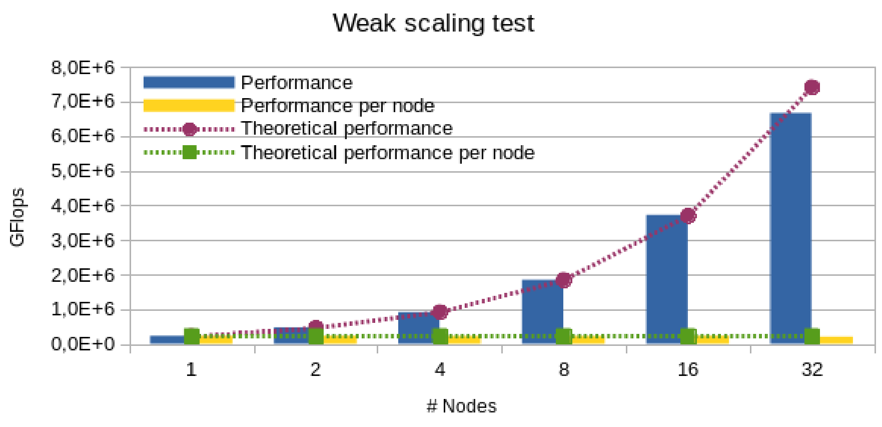

Figure 12: Weak scaling test 4

Therefore, despite the fact that the execution with inside-outside mixed MPI has presented a small delay, this is not of a great magnitude. In addition, this delay may occur only as a consequence of some uncontrolled noise. Therefore, we can say that the **execution with inside-outside mixed MPI is a good solution in terms of computational time, although it may not be the best.**

12 Discussion

Achieving the end of this document, we have obtained some interesting results, so it is important to remember and summarize them.

The main goal of this study was to obtain a possible environment solution which allow to evolve and improve sophisticated HPC scientific software with all its dependencies at a faster pace than using traditional software integration and deployment.

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. Nowadays, continuous integration, agile development methodologies and rolling releases are concepts being used in modern scientific software development. Having an HPC software up-to-date in this scenario and with traditional integration and deployment becomes a huge bottleneck. Therefore, the ultimate goal has been to reduce as far as possible these drawbacks of conventional deployment and to achieve a solution as flexible as possible.

After observing the possible realizable solutions, it was concluded that the one that best adapts for this concrete case consists on a containers-based system. The main issue of the selected containerization technology, Singularity, relies on its integration with Open MPI. Most of the work in this document is an effort to find a generic way to deploy whatever MPI parallel application within a container on Finis Terrae II.

In order to achieve the goal, different approaches were proposed, since the expected response was not initially obtained. Once a possible solution was reached, efforts were made to obtain the maximum possible stability of the same, carrying out numerous tests, as well as executions of real and complex applications. After multiple test-error executions, we identified those parameters that influenced the correct functioning of the environment, such as libraries involved in MPI communication or the use of different compilers. Once these parameters were identified, the environment deployment was fully automated through scripts to obtain stable environments under which it is possible to execute the desired applications.

In spite of having made many advances with the "Binding libraries and their dependencies" approach, it should be remembered that this is not the only approach that we have detected throughout this study. Any of these approaches can be applicable as long as the objective are achieved. We will review the different characteristics of each of these approaches, analyzing their pros and cons.

12.1 "Use exactly the same Open MPI versions" approach

It was the first proved 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, Open MPI version must match exactly within the container and the host.

- Pros
    1. Containers' philosophy is not corrupted.
    2. We can ensure 100% ABI compatibility.
- Cons

1. 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.
2. *Srun* process manager is not compatible. As the container's employed process manager is *mpirun*, it conflicts with the host's *srun* process manager, making impossible their mix. We are forced to use mpirun if we need parallel communication.
3. 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.
4. Container must have installed some libraries in order to use some HPC resources (libiverbs, ibutils, etc.)
5. Inexistent interoperability.

12.2 "Binding MPI libraries and their dependencies" approach

This is the approach we have developed most throughout this study. It consists on prepend the Open MPI libraries inside the container, as well as their dependencies, with the Open MPI libraries installed in the host, making use of the Singularity binding option. Remember several changes happen when the Open MPI version changed from 1.X to 2.X, so we need at least two Open MPI versions installed and configured at the host: a 1.10 version or lower (not less than 1.6.5), and a 2.X version. Every time the major version changes, the system administrator must install and configure an Open MPI with this new major version, and deploy an environment to the new containers that make use of these new versions.

- Pros

  1. Medium system administrator iteration. The system administrator must install and configure at least one Open MPI version per major version available.
  2. We can use the *srun* process manager as we are using exactly the same PMI. This process manager is the recommended one to use at the Finis Terrae II.
  3. High interoperability level.
  4. The container does not need to have installed specific hardware related libraries.

- Cons

  1. 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.
  2. In general, we can not ensure 100% ABI compatibility. Reader can read the ABI tracker page if he wants to know more.
  3. 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.

4. All containers must go through a bootstrap process to make their environment suitable for our configuration (create a series of folders, make links, etc.)

Finally, once this environment was obtained, it was tested. For this purpose, critical parameters were analyzed to determine if the solution reached a viable alternative to run the software under the Finis Terrae II cluster. Thus, 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.

13 Future work

13.1 PMIx approach

Another possible solution to the Open MPI interoperability, and therefore to the portability of distributed applications with Singularity, has been identified. Under this approach, we will make use of PMIx as process manager at the host and within the container.

The solution is promising, but at the time of this study the technologies are not mature enough. The expected right functioning of these technologies, for our goals, will be ready with Open MPI v.3 and PMIx v.2.1. These versions are not already released.

In addition, PMIx is also not supported by *SLURM* version, the resource manager and process manager, currently available and recommended in the Finis Terrae II.

14 Annexes

This final section contains most of the code employed throughout this study, which can be useful to the reader.

14.1 Bootstrap definition file

The following bootstrap definition file must be called with a Singularity container image in order to create a container with the latest stable Ubuntu repo available on the Docker Hub. At the moment this lines are been written this is Ubuntu 16.04.2 LTS (Ubuntu Xenial). This container will install Open MPI, downloading it from its official page (not from repositories). In this concrete case, it will be installed ompi/1.10.2, but simply change the version number on the necessary lines to download and configure another version.

To do that, simply use the Singularity bootstrap option. It is important to use it with superuser permissions:

Listing 14: Employing Singularity bootstrap option

```
sudo singularity bootstrap container.img bootstrap.def
```

Listing 15: Bootstrap definition file to create an Ubuntu container with Open MPI installed and configured

```
BootStrap: docker
From: ubuntu
IncludeCmd: yes

%post
        # Get the system ready
        sed -i 's/main/main restricted universe/g' /
           ↪ etc/apt/sources.list
        apt-get update
        apt-get install -y --no-install-recommends apt
           ↪ -utils bash build-essential curl gcc git
           ↪  python python-pip pkg-config python-dev
           ↪  python-setuptools rsync software-
           ↪ properties-common time unzip vim nano
           ↪ wget zip
        # Need to be split in 2 different apts
        apt-get install -y --no-install-recommends
           ↪ zlib1g-dev libopenblas-dev libc6-dev
           ↪ libcurl4-openssl-dev libfreetype6-dev
           ↪ libgcc-5-dev libpng-dev libzmq3-dev
        apt-get clean
        # Update to the latest pip (newer than repo)
        pip install --no-cache-dir --upgrade pip
        # Install other commonly-needed packages
        pip install --no-cache-dir --upgrade future
           ↪ matplotlib scipy
```

```
# IB stuff, based on https://community.
   ↪ mellanox.com/docs/DOC-2431
apt-get install -y dkms infiniband-diags
   ↪ libibverbs* ibacm librdmacm* libmlx4*
   ↪ libmlx5* mstflint libibcm.* libibmad.*
   ↪ libibumad* opensm srptools libmlx4-dev
   ↪ librdmacm-dev rdmacm-utils ibverbs-utils
   ↪  perftest vlan ibutils
apt-get install -y libtool autoconf automake
   ↪ build-essential ibutils ibverbs-utils
   ↪ rdmacm-utils infiniband-diags perftest
   ↪ librdmacm-dev libibverbs-dev libmlx4-1
   ↪ numactl libnuma-dev autoconf automake
   ↪ gcc g++ git libtool pkg-config
apt-get install -y libnl-3-200 libnl-route
   ↪ -3-200 libnl-route-3-dev libnl-utils
# Open MPI dependencies
apt-get install -y libopenmpi-dev openmpi-
   ↪ common openmpi-bin openmpi-doc dapl2-
   ↪ utils libdapl-dev libdapl2 libibverbs1
   ↪ libibverbs-dev librdmacm1 libcxgb3-1
   ↪ libipathverbs1 libmlx4-1 libmlx5-1
   ↪ libmthca1 libnes1 libpmi0 libpmi0-dev
   ↪ libslurm29 libslurm-dev libsysfs2
   ↪ libsysfs-dev
# Install Open MPI
cd /tmp/
# IMPORTANT!: Change the Open MPI version to
   ↪ the desired one
wget https://www.open-mpi.org/software/ompi/v1
   ↪ .10/downloads/openmpi-1.10.2.tar.gz -O
   ↪ openmpi-1.10.2.tar.gz
tar -xzf openmpi-1.10.2.tar.gz
mkdir -p /tmp/openmpi-1.10.2/build
cd /tmp/openmpi-1.10.2/build
../configure --prefix=/usr --with-slurm --
   ↪ enable-shared --enable-mpi-thread-
   ↪ multiple --with-verbs --enable-mpirun-
   ↪ prefix-by-default --with-hwloc --disable
   ↪ -dlopen --with-pmi --with-pmi-libdir=/
   ↪ usr/lib/x86_64-linux-gnu
make clean
make all install
export LD_LIBRARY_PATH=/usr/local/lib:
# Create some folders
mkdir -p /host/mpi /scratch /tmp /home /opt/
   ↪ cesga /mnt /usr/local/openmpi /usr/lib64
   ↪ /slurm /opt/mellanox /host/lib64 /host/
   ↪ filtered/lib64 /host/usr/lib64 /host/
   ↪ filtered/usr/lib64 /etc/libibverbs.d /
```

```
    ↪ run/munge /etc/slurm /.singularity.d/
    ↪ libs
# Last
echo "driver mlx4" > /etc/libibverbs.d/mlx4.
    ↪ driver
echo "driver mlx5" > /etc/libibverbs.d/mlx5.
    ↪ driver
adduser slurm || echo "User exists"
cat /etc/usr_lib64.txt | xargs -I{} ln -fs /
    ↪ host/usr/lib64/{} /host/filtered/usr/
    ↪ lib64/{}
cat /etc/lib64.txt | xargs -I{} ln -fs /host/
    ↪ lib64/{} /host/filtered/lib64/{}
```

14.2 Generic calling script

To have a way to reserve resources, call and execute multiple executions of the same program, in order to be able to execute the possible combinations, a bash script was written that automated the whole process. Although it may seem long, we only have to consider these aspects:

- There are three distinct areas
  1. **DEBUGGING MODE**: If we call the script with the `--debug` option, it will print on the screen the employed compilers as well as the path where the host libraries are ubicated.
  2. **FUNCTIONS**: It only contains a function which shows in a more readable way the separation between the tests.
  3. **MAIN**: The script itself. It is responsible for correctly fitting all possible combinations of Open MPI versions inside and outside the container as well as the available external compilers. It is a little complicated because in Finis Terrae II the software installation have not always been done in the same way. Thus, Open MPI versions 2.0.2 and 2.1.1 made use of Easybuild, so its installation directory changes substantially.
- This script is strongly dependent on the sbatchs scripts, the ones really in charge of launching the processes in the cluster. Because of the use of variables, one has no meaning without the other and vice versa.

Listing 16: Generic calling script

```
#!/bin/bash

#/*---- DEBUGGING MODE ----*/
DEBUG=false
if [ $# -eq 1 ]; then
   for arg in "$@"; do
      if [ "$arg" == "--debug" ]; then
         DEBUG=true
      fi
   done
fi


#/*---- FUNCTIONS ----*/
printfCenter() {
   termwidth="$(tput cols)"
   padding="$(printf '%0.1s' ={1..500})"
   printf '%*.*s %s %*.*s\n' 0 "$(((termwidth-2-${#1})
      ↪ /2))" "$padding" "$1" 0 "$(((termwidth-1-$
      ↪ {#1})/2))" "$padding"
}


#/*---- MAIN ----*/
export HOST_LIBS=/path/to/host-libs/
CONTAINER_OPEN_MPI_VERSIONS=( "1.10.2" "2.0.0" "2.0.1"
   ↪  "2.0.2" "2.1.1" )

# NO EASYBUILD VERSION ( 1.10.2 | 2.0.0 | 2.0.1 )
OPEN_MPI_VERSIONS=( "1.10.2" "2.0.0" "2.0.1" )
COMPILER=gcc; export COMPILER=$COMPILER
for OPEN_MPI_VERSION in "${OPEN_MPI_VERSIONS[@]}"; do
   export OPEN_MPI_VERSION=$OPEN_MPI_VERSION
   if [ "$OPEN_MPI_VERSION" = "1.10.2" ]; then
      export LIBS_PATH=/host/filtered/lib64:/host/
         ↪ filtered/usr/lib64:/host/mpi/lib/ompi_1.X
   else
      export LIBS_PATH=/host/filtered/lib64:/host/
         ↪ filtered/usr/lib64:/host/mpi/lib/ompi_2.X
   fi
   if [ "$OPEN_MPI_VERSION" = "2.0.1" ]; then
      COMPILER_VERSION=6.3.0
   else
      COMPILER_VERSION=6.1.0
   fi
   export COMPILER_VERSION=$COMPILER_VERSION
   module load $COMPILER/$COMPILER_VERSION openmpi/
      ↪ $OPEN_MPI_VERSION singularity
   for CONTAINER_OPEN_MPI_VERSION in "${
      ↪ CONTAINER_OPEN_MPI_VERSIONS[@]}"; do
      export CONTAINER_OPEN_MPI_VERSION=
         ↪ $CONTAINER_OPEN_MPI_VERSION
      printfCenter "START! FT2: $OPEN_MPI_VERSION |
         ↪ Container: $CONTAINER_OPEN_MPI_VERSION"
      if [ "$DEBUG" == "true" ]; then
         echo COMPILER=$COMPILER/$COMPILER_VERSION
         echo LIBS_PATH=$LIBS_PATH
      else
         sbatch sbatch_script_1.sh
```

```
        fi
    done
done


# EASYBUILD VERSION ( 2.0.2 | 2.1.1 )
OPEN_MPI_VERSIONS=( "2.0.2" "2.1.1" )
export LIBS_PATH=/host/filtered/lib64:/host/filtered/
    ↪ usr/lib64:/host/mpi/lib/ompi_2.X
for OPEN_MPI_VERSION in "${OPEN_MPI_VERSIONS[@]}"; do
    export OPEN_MPI_VERSION=$OPEN_MPI_VERSION
    if [ "$OPEN_MPI_VERSION" = "2.0.2" ]; then
        COMPILER=gcc
        COMPILER_VERSION=6.3.0
        COMPILER_EASYBUILD=GCC
        COMPILER_VERSION_EASYBUILD=6.3.0-2.27
    else
        COMPILER=intel
        COMPILER_VERSION=2016
        COMPILER_EASYBUILD=iccifort
        COMPILER_VERSION_EASYBUILD=2016eb
    fi
    export COMPILER_EASYBUILD=$COMPILER_EASYBUILD
    export COMPILER_VERSION_EASYBUILD=
        ↪ $COMPILER_VERSION_EASYBUILD
    module load $COMPILER/$COMPILER_VERSION openmpi/
        ↪ $OPEN_MPI_VERSION singularity
    for CONTAINER_OPEN_MPI_VERSION in "${
        ↪ CONTAINER_OPEN_MPI_VERSIONS[@]}"; do
        export CONTAINER_OPEN_MPI_VERSION=
            ↪ $CONTAINER_OPEN_MPI_VERSION
        printfCenter "START! FT2: $OPEN_MPI_VERSION |
            ↪ Container: $CONTAINER_OPEN_MPI_VERSION"
        if [ "$DEBUG" == "true" ]; then
            echo COMPILER=$COMPILER/$COMPILER_VERSION
            echo LIBS_PATH=$LIBS_PATH
        else
            sbatch sbatch_script_2.sh
        fi
    done
done
```

14.3 Sbatch scripts

The following scripts are 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 17: Sbatch script 1

```
#!/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 18: Sbatch script 2

```
#!/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/easybuild/software/Open MPI/
   ↪ $OPEN_MPI_VERSION-$COMPILER_EASYBUILD-
   ↪ $COMPILER_VERSION_EASYBUILD/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"
```

14.4 Host binded and filtered libraries

At this final subsection, we can observe the final filtered list of all the libraries needed to be binded inside the container from the host.

Listing 19: Host filtered libraries

```
/host/filtered/
|-- lib64
|   |-- libnl.so.1 -> /host/lib64/libnl.so.1
|   |-- libnl.so.1.1.4 -> /host/lib64/libnl.so.1.1.4
|   |-- libpci.so.3 -> /host/lib64/libpci.so.3
|   |-- libpci.so.3.1.10 -> /host/lib64/libpci.so
   ↪ .3.1.10
|   |-- libslurm.la -> /host/usr/lib64/libslurm.la
|   |-- libslurm.so -> /host/usr/lib64/libslurm.so
|   |-- libslurm.so.28 -> /host/usr/lib64/libslurm.so
   ↪ .28
|   |-- libslurm.so.28.0.0 -> /host/usr/lib64/libslurm
   ↪ .so.28.0.0
|   |-- libslurmdb.la -> /host/usr/lib64/libslurmdb.la
|   |-- libslurmdb.so -> /host/usr/lib64/libslurmdb.so
|   |-- libslurmdb.so.28 -> /host/usr/lib64/libslurmdb
   ↪ .so.28
|   `-- libslurmdb.so.28.0.0 -> /host/usr/lib64/
   ↪ libslurmdb.so.28.0.0
`-- usr
    `-- lib64
        |-- libhwloc.so -> /host/usr/lib64/libhwloc.so
        |-- libhwloc.so.1 -> /host/usr/lib64/libhwloc.
           ↪ so.1
        |-- libhwloc.so.1.0.1 -> /host/usr/lib64/
           ↪ libhwloc.so.1.0.1
        |-- libhwloc.so.5.1.0 -> /host/usr/lib64/
           ↪ libhwloc.so.5.1.0
        |-- libhwloc.so.5.7.2
        |-- libhwloc.so.5.bkp -> /host/usr/lib64/
           ↪ libhwloc.so.5
        |-- libibcm.a -> /host/usr/lib64/libibcm.a
        |-- libibcm.so -> /host/usr/lib64/libibcm.so
        |-- libibcm.so.1 -> /host/usr/lib64/libibcm.so
           ↪ .1
        |-- libibcm.so.1.0.0 -> /host/usr/lib64/
           ↪ libibcm.so.1.0.0
        |-- libibdm.a -> /host/usr/lib64/libibdm.a
        |-- libibdm.so -> /host/usr/lib64/libibdm.so
        |-- libibdm.so.1 -> /host/usr/lib64/libibdm.so
            ↪ .1
        |-- libibdm.so.1.1.1 -> /host/usr/lib64/
           ↪ libibdm.so.1.1.1
        |-- libibdmcom.a -> /host/usr/lib64/libibdmcom
           ↪ .a
        |-- libibdmcom.so -> /host/usr/lib64/
           ↪ libibdmcom.so
        |-- libibdmcom.so.1 -> /host/usr/lib64/
           ↪ libibdmcom.so.1
        |-- libibdmcom.so.1.1.1 -> /host/usr/lib64/
           ↪ libibdmcom.so.1.1.1
        |-- libiberty.a -> /host/usr/lib64/libiberty.a
        |-- libibmad.so -> /host/usr/lib64/libibmad.so
        |-- libibmad.so.5 -> /host/usr/lib64/libibmad.
           ↪ so.5
        |-- libibmad.so.5.5.0 -> /host/usr/lib64/
           ↪ libibmad.so.5.5.0
        |-- libibmscli.a -> /host/usr/lib64/libibmscli
           ↪ .a
        |-- libibmscli.so -> /host/usr/lib64/
           ↪ libibmscli.so
        |-- libibmscli.so.1 -> /host/usr/lib64/
           ↪ libibmscli.so.1
        |-- libibmscli.so.1.0.0 -> /host/usr/lib64/
           ↪ libibmscli.so.1.0.0
        |-- libibnetdisc.a -> /host/usr/lib64/
           ↪ libibnetdisc.a
        |-- libibnetdisc.so -> /host/usr/lib64/
           ↪ libibnetdisc.so
        |-- libibnetdisc.so.5 -> /host/usr/lib64/
           ↪ libibnetdisc.so.5
        |-- libibnetdisc.so.5.3.0 -> /host/usr/lib64/
           ↪ libibnetdisc.so.5.3.0
        |-- libibsysapi.a -> /host/usr/lib64/
           ↪ libibsysapi.a
        |-- libibsysapi.so -> /host/usr/lib64/
           ↪ libibsysapi.so
        |-- libibsysapi.so.1 -> /host/usr/lib64/
           ↪ libibsysapi.so.1
        |-- libibsysapi.so.1.0.0 -> /host/usr/lib64/
           ↪ libibsysapi.so.1.0.0
        |-- libibumad.so -> /host/usr/lib64/libibumad.
           ↪ so
        |-- libibumad.so.3 -> /host/usr/lib64/
           ↪ libibumad.so.3
        |-- libibumad.so.3.1.0 -> /host/usr/lib64/
           ↪ libibumad.so.3.1.0
```

```
|-- libibverbs.so -> /host/usr/lib64/
   ↪ libibverbs.so
|-- libibverbs.so.1 -> /host/usr/lib64/
   ↪ libibverbs.so.1
|-- libibverbs.so.1.0.0 -> /host/usr/lib64/
   ↪ libibverbs.so.1.0.0
|-- liblustreapi.a -> /host/usr/lib64/
   ↪ liblustreapi.a
|-- liblustreapi.so -> /host/usr/lib64/
   ↪ liblustreapi.so
|-- libmlx4-rdmav2.so -> /host/usr/lib64/
   ↪ libmlx4-rdmav2.so
|-- libmlx4.a -> /host/usr/lib64/libmlx4.a
|-- libmlx5-rdmav2.so -> /host/usr/lib64/
   ↪ libmlx5-rdmav2.so
|-- libmlx5.a -> /host/usr/lib64/libmlx5.a
|-- libmthca-rdmav2.so -> /host/usr/lib64/
   ↪ libmthca-rdmav2.so
|-- libmunge.so.2 -> /host/usr/lib64/libmunge.
   ↪ so.2
|-- libmunge.so.2.0.0 -> /host/usr/lib64/
   ↪ libmunge.so.2.0.0
|-- libnuma.a -> /host/usr/lib64/libnuma.a
|-- libnuma.so -> /host/usr/lib64/libnuma.so
|-- libnuma.so.1 -> /host/usr/lib64/libnuma.so
   ↪ .1
|-- libpmi.la -> /host/usr/lib64/libpmi.la
|-- libpmi.so -> /host/usr/lib64/libpmi.so
|-- libpmi.so.0 -> /host/usr/lib64/libpmi.so.0
|-- libpmi.so.0.0.0 -> /host/usr/lib64/libpmi.
   ↪ so.0.0.0
|-- libpmi2.la -> /host/usr/lib64/libpmi2.la
|-- libpmi2.so -> /host/usr/lib64/libpmi2.so
|-- libpmi2.so.0 -> /host/usr/lib64/libpmi2.so
   ↪ .0
|-- libpmi2.so.0.0.0 -> /host/usr/lib64/
   ↪ libpmi2.so.0.0.0
|-- librdmacm.a -> /host/usr/lib64/librdmacm.a
|-- librdmacm.so -> /host/usr/lib64/librdmacm.
   ↪ so
|-- librdmacm.so.1 -> /host/usr/lib64/
   ↪ librdmacm.so.1
|-- librdmacm.so.1.0.0 -> /host/usr/lib64/
   ↪ librdmacm.so.1.0.0
|-- libxml2.a -> /host/usr/lib64/libxml2.a
|-- libxml2.so -> /host/usr/lib64/libxml2.so
|-- libxml2.so.2 -> /host/usr/lib64/libxml2.so
   ↪ .2
|-- libxml2.so.2.7.6 -> /host/usr/lib64/
   ↪ libxml2.so.2.7.6
```

```
‘-- slurm -> /host/usr/lib64/slurm
```

References

[1] MSO4SC: D3.1 Detailed Specifications for the Infrastructure, Cloud Management and MSO Portal [Internet] - Cemosis: Centre for modeling and simulation in Strasbourg [Quoted August 28, 2017]. Recovered from: <http://book.mso4sc.cemosis.fr/deliverables/d3.1/>

[2] MSO4SC: CESGA Singularity 2.2.1 Manual v0.3 - Fundación Pública Galega Centro Tecnolóxico de Supercomputación de Galicia (CESGA) [Quoted August 28, 2017].

[3] Horizon 2020 - The EU Framework Programme for Research and Innovation [Internet] [Quoted August 03, 2017]. Recovered from: <https://ec.europa.eu/programmes/horizon2020/>

[4] Easybuild [Internet] - EasyBuild: building software with ease. [Quoted August 03, 2017]. Recovered from: <https://easybuilders.github.io/easybuild/>

[5] Docker Hub [Internet] [Quoted August 04, 2017]. Recovered from: <https://hub.docker.com/>

[6] Singularity Hub [Internet] [Quoted August 04, 2017]. Recovered from: <https://singularityhub.com/>

[7] Singularity on HPC [Internet] - Singularity [Quoted August 04, 2017]. Recovered from: <http://singularity.lbl.gov/docs-hpc>

[8] Process Management Interface Exascale Github Wiki [Internet] - Github [Quoted August 29, 2017]. Recovered from: <https://github.com/pmix/pmix/wiki>

[9] ring_c.c code [Internet] - Open MPI (open-mpi) Github repository [Quoted August 10, 2017]. Recovered from: <https://raw.githubusercontent.com/open-mpi/ompi/master/examples/ring_c.c>

[10] Intel® MPI Library [Internet] - Intel Developer Zone [Quoted August 10, 2017]. Recovered from: <https://software.intel.com/en-us/intel-mpi-library>

[11] Slurm Workload Manager Documentation [Internet] - Slurm Workload Manager [Quoted August 11, 2017]. Recovered from: <https://slurm.schedmd.com/>

[12] Using Host libraries: GPU drivers and Open MPI BTLs [Internet] - Singularity [Quoted August 11, 2017]. Recovered from: <http://singularity.lbl.gov/tutorial-gpu-drivers-open-mpi-mtls>

[13] Open MPI API/ABI changes review [Internet] - ABI tracker project [Quoted August 25, 2017]. Recovered from: <https://abi-laboratory.pro/tracker/timeline/openmpi/>

[14] MPI for Python (MPI4Py) Official Page [Internet] [Quoted August 23, 2017]. Recovered from: mpi4py.scipy.org/

[15] Smith-Waterman algorithm [Internet] - University of Southern California [Quoted August 23, 2017]. Recovered from: <http://dornsife.usc.edu/assets/sites/516/docs/papers/msw_papers/msw-042.pdf>

[16] Smith-Waterman parallel adaptation code [Internet] - Center for Research Computing, University of Pittsburgh [Quoted August 23, 2017]. Recovered from: <http://coco.sam.pitt.edu/~emeneses/wp-content/uploads/2013/11/code.zip>

[17] OSU Micro Benchmarks [Internet] - MVAPICH: MPI over InfiniBand, Omni-Path, Ethernet/iWARP, and RoCE [Quoted July 24, 2017]. Recovered from: <http://mvapich.cse.ohio-state.edu/userguide/virt/#_osu_micro_benchmarks>

[18] OSU Micro Benchmarks README file [Internet] - MVAPICH: MPI over InfiniBand, Omni-Path, Ethernet/iWARP, and RoCE [Quoted July 24, 2017]. Recovered from: <http://mvapich.cse.ohio-state.edu/static/media/mvapich/README-OMB.txt>

[19] HPC work-flow with MPI Parallel/Distributed jobs (OSU Microbenchmarks, HPL) [Internet] - University of Luxembourg High Performance Computing (HPC) Team [Quoted August 03, 2017]. Recovered from: <https://ulhpc-tutorials.readthedocs.io/en/latest/advanced/OSU_MicroBenchmarks/>

[20] Some Micro-benchmarks for HPC: 10Gb Ethernet on EC2 vs QDR InfiniBand [Internet] - Thinking out loud, personal Adam DeConinck blog [Quoted August 03, 2017]. Recovered from: <https://blog.ajdecon.org/some-micro0-benchmarks-for-hpc-10gb-ethernet-o/>

[21] Implementation and comparison of RDMA over Ethernet [Internet] - National Security Education Center - Los Alamos National Laboratory [Quoted August 03, 2017]. Recovered from: <http://www.lanl.gov/projects/national-security-education-center/information-science-technology/_assets/docs/2010-si-docs/Team_CYAN_Implementation_and_Comparison_of_RDMA_Over_Ethernet_Presentation.pdf>

[22] STREAM: Sustainable Memory Bandwidth in High Performance Computers [Internet] [Quoted July 28, 2017]. Recovered from: <https://www.cs.virginia.edu/stream/>

[23] STREAM ref. - STREAM: Sustainable Memory Bandwidth in High Performance Computers [Internet] [Quoted July 28, 2017]. Recovered from: <https://www.cs.virginia.edu/stream/ref.html>

[24] HPL - A Portable Implementation of the High-Performance Linpack Benchmark for Distributed-Memory Computers [Internet] - The Innovative Computing Laboratory (ICL) of the University of Tennesse [Quoted July 28, 2017]. Recovered from: <http://www.netlib.org/benchmark/hpl/>