# Profiling HPC applications in containerized environments

**Albert Sanuy Lostes**
Master's degree in Computer Engineering
High Performance Computing area

**Sergio Iserte Agut**
**Josep Jorba Esteve**

February 2022

## FINAL PROJECT SHEET

| | |
|---|---|
| **Project title:** | Profiling HPC applications in containerized environments |
| **Author name:** | Albert Sanuy Lostes |
| **Consultant name:** | Sergio Iserte Agut |
| **Tutor name:** | Josep Jorba Esteve |
| **Delivery date (mm/yyyy):** | 06/2022 |
| **Work area:** | High Performance Computing |
| **University degree:** | Master's degree in Computer Engineering |
| **Key words:** | Cluster Computing, Profiling, Containers, Scientific Applications |

**Abstract:**

Scientific studies very often rely on supercomputers to solve difficult problems. However, reproducibility is one of the core principles in any scientific research and it is often expected that the findings of any study can be replicated with a high degree of reliability. Running these applications in containers that have been prepared and curated beforehand can provide the desired reliability, remove unnecessary complexity and reduce the manual interaction to reduce the risk of human errors.

The overall goal of this project is to use Docker to create an image that contains a distributed scientific application in addition to the necessary tools that allow profiling the behaviour of the program after its execution with different workloads. The Docker containers will be managed with Singularity , the most widely used container system for HPC.

The infrastructure of the HPC cluster used for this study is composed of Raspberries Pi 4 Model B hosted on-premises.

## INDEX

**1. Introduction ........ 1**

1.1. Project motivation ........ 1

1.2. Goals and objectives ........ 2

1.3. Approach and method followed ........ 3

1.4. Work breakdown ........ 4

**2. Infrastructure ........ 6**

2.1. Architecture Overview ........ 6

2.2. Raspberry model ........ 7

2.3. Network configuration ........ 8

2.4. Operating system ........ 10

**3. Virtualization ........ 11**

3.1. Virtual machines ........ 11

3.2. Containers ........ 11

**4. Libraries overview ........ 13**

4.1. OpenMPI ........ 13

4.2. OpenFOAM ........ 13

4.3. Extrae ........ 14

4.4. Paraver ........ 14

**5. Libraries compilation ........ 15**

5.1. Context ........ 15

5.2. Compile OpenMPI ........ 16

5.3. Compile OpenFOAM ........ 17

5.4. Compile Extrae ........ 19

5.5. Compile Paraver ........ 22

**6. Docker ........ 23**

6.1. Dockerfile ........ 23

6.2. Build image ........ 25

6.3. Publish image ........ 26

**7. Singularity .......... 27**

7.1. Installation .......... 27

7.2. Singularity Image File .......... 28

**8. Running the container .......... 29**

8.1. Case study .......... 29

8.2. Execution .......... 30

8.3. Extrae configuration file .......... 33

8.4. Analysis .......... 35

**9. Conclusions .......... 39**

**10. Bibliography .......... 41**

# 1. INTRODUCTION

## 1.1. Project motivation

High performance computing, hereinafter referred to as HPC, has been the foundation for many scientific, industrial, and societal advancements since the second third of the twentieth century to this day.

The reason HPC has been crucial in many investigations is because it gives the ability to process data and perform complex computations at very high speeds in many situations that otherwise would not be feasible, or the results would take so long that would make them useless.

One of the best-known types of HPC solutions is the supercomputers. A supercomputer usually consists of hundreds or thousands of compute nodes that work together to process data in parallel and complete small jobs that are part of a more complex task. The collection of compute nodes often is referred to as an HPC cluster.

Nowadays, it is a frequent practice for scientific studies to rely on these clusters to solve difficult problems, however, in the recent decades, many published scientific results failed because they were difficult or even impossible to reproduce. Reproducibility is indeed one of the core principles in any scientific research and often it is expected that the findings of any study can be replicated with a high degree of reliability.

In fact, this has not only been a challenge in academic investigations but also in other fields such as software engineering, contributing to the development of tools that could provide an abstraction to the machine-specific settings. One of the most popular software solutions to achieve that level of isolation is Docker[1].

In this context, the overall goal of this project is to use Docker to create an image that contains a distributed scientific application in addition to the necessary tools that allow profiling the behaviour of the program after its execution. The Docker containers will be managed with Singularity[2], the most widely used container system for HPC.

It is worth noting that the infrastructure of the HPC cluster used for this study is composed of Raspberries Pi 4 Model B[3] hosted on-premises. Both the tools and the infrastructure will be detailed in later chapters.

[1] Link to https://www.docker.com
[2] Link to https://sylabs.io/singularity
[3] Link to https://www.raspberrypi.com

## 1.2. Goals and objectives

The ultimate goal of this project is to use Singularity in a cluster to simultaneously deploy and run Docker containers containing an HPC application and use some post-processing tools to generate trace-files for a post-mortem analysis that helps us understand how the microprocessor and other components of the system were behaving during the execution.

In order to achieve that goal, there are four important milestones that need to be completed:

1. **Familiarise with the ecosystem:** Includes understanding the architecture of the HPC cluster, the libraries and tools needed in order to conduct this study and what are some of the most popular alternatives in the market.
2. **Create an image:** Build the image that contains the HPC application that will be profiled along with the necessary tools to generate the traces and analyse them.
3. **Run the containers in the cluster:** Use Singularity as a container system to deploy and run in the cluster Docker containers using the image built.
4. **Profile the HPC application:** Use one post-processing tool for a post-mortem analysis of the HPC application that ran in each container. Investigate the impact of running different workloads in the cluster.

Similarly, each milestone can be broken down into different objectives which are described in the following table.

| Milestone | Objective |
|---|---|
| Familiarise with the ecosystem | Understand the infrastructure of the cluster used in the study of this project. Given the nature of the system, it is important to identify the limitations and constraints of the architecture. |
| | Learn about MPI[4], the Message Passing Interface-programming paradigm used to perform the parallel processing in the simulation of the HPC application. |
| | Become familiar with an HPC application that leverages the MPI standard. Preferably, a distributed scientific application. |
| | Read about different post-processing tools available to generate trace-files of the execution of an HPC application for a post-mortem analysis and choose the most appropriate libraries for this study. |

[4] Link to https://www.mpi-forum.org

| | |
|---|---|
| | Describe what are the advantages of Docker over other heavier weight virtualisation alternatives. |
| **Create an image** | Compile the selected HPC application, some implementation of the MPI standard and the necessary post-processing tools in order to generate the binary files that correspond to the architecture of the HPC cluster. |
| | Implement a Docker image using a Linux-based operating system that includes the binary files resulting from the compilation and that will serve to profile the system. |
| **Run the containers in the cluster** | Become familiar with Singularity, the container platform used to deploy and manage the lifecycle of the containers running in the cluster. |
| | Use Singularity to deploy multiple Docker containers simultaneously in the cluster. |
| **Profile the HPC application** | Analyse and understand the behaviour of the system with different workloads using profiling tools. |

Table 1. Description of the milestones and the objectives

## 1.3. Approach and method followed

The scope of this project can be divided into three important pieces of work that must be executed sequentially in the following order.

First, it is necessary to investigate and figure out the infrastructure of the HPC cluster that will serve as a pseudo supercomputer for this project. Understanding the basics of the system, different nodes that compose it, the role they play, how they intercommunicate each other and any other necessary component is essential before starting the actual work.

Afterwards, it is required a thorough comprehension of the different libraries that will be used both for running the simulation and post-processing the traces to effectively gather relevant information during the research. An important part of the work at this stage involves building the Docker image and managing the lifecycle of the containers using Singularity.

Finally, it must be put in context all the details collected throughout the process in order to extract conclusions, list unexpected pitfalls and describe the behaviour of the simulations across different workloads.

## 1.4. Work breakdown

According to the course plan, the estimated dedication of the student for the Master's Final Project is three hundred hours and has a duration of one hundred and twenty-two days from the beginning of the semester until the last deliverable.

Taking these figures into account, below is a Gantt chart with the time planning of these tasks to fulfil the objectives outlined in the previous section.

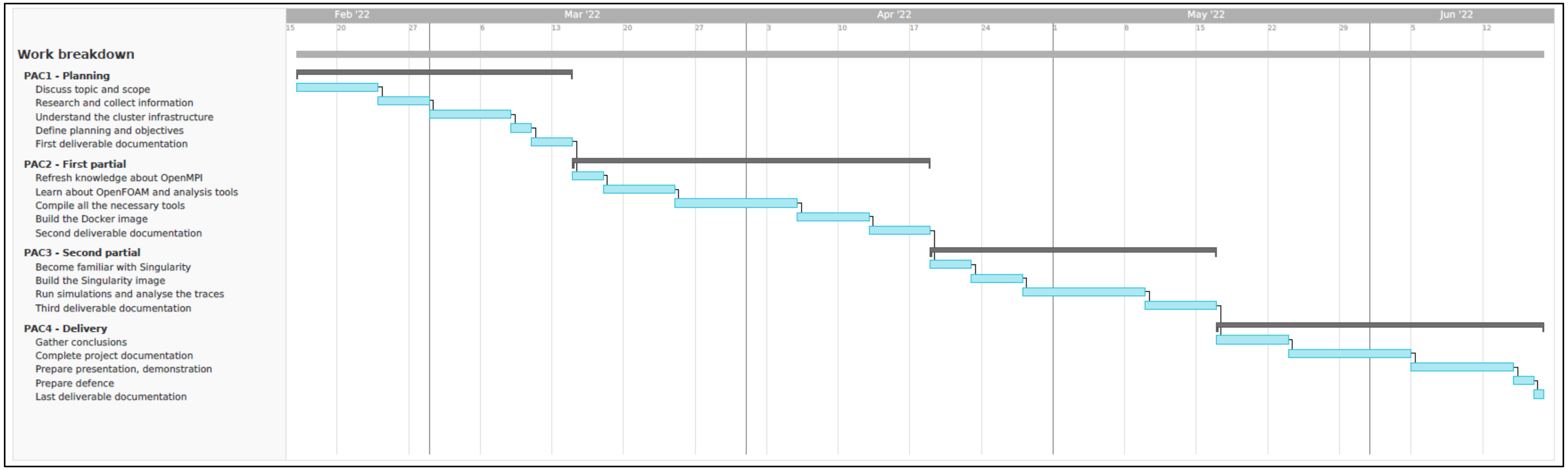


Figure 1. Work breakdown in a Gantt chart

# 2. INFRASTRUCTURE

## 2.1. Architecture Overview

The HPC cluster uses two Raspberries Pi 4 Model B that will emulate a supercomputer on a small-scale. The cluster is composed of two nodes, each of them mapping to a single underlying physical Raspberry Pi, which are connected by a local area network to communicate with each other to run executions in parallel and share the intermediate calculations in a network file system.

The rationale for having more than one Raspberry, is that one acts as the *front-end*, which is the node used to connect from an external host to schedule a job to the cluster, whereas the rest of the Raspberry act as the *workers*, which are the nodes that are responsible for the all the computation.

It is worth noting that for this study, considering that only two Raspberries are available, the front-end node will also be used as a worker node, although that is not the recommended practice in an actual supercomputer running in production.

Each of the Raspberries is connected to a power supply, which supplies the electric power, and to a switch via a RJ45 Ethernet connector. The switch, a TP-LINK TL-SG1005D with 5 ports available, acts as the network device in an actual HPC cluster, handling the intercommunication between the two devices. In addition, the switch is connected to a router that allows the communication between the rest of the devices (e.g., some host) and provides access to the internet.

The following figure outlines the architecture of the system containing the front-end, one worker, the switch and the router.

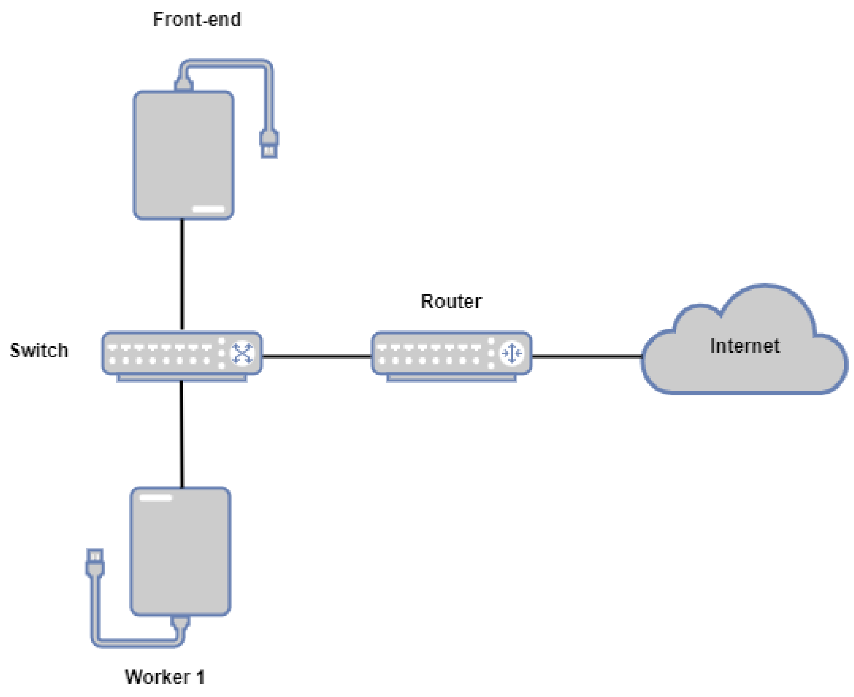


Figure 2. Architecture of the cluster

In broad terms, the key difference between a switch and a router is that a switch works on the *data link layer* of the OSI model (Open Systems Interconnection Model) and connects different devices (e.g., the nodes, some hosts), whereas a router works on the *network layer* of the OSI model and connects different networks.

## 2.2. Raspberry model

The model of the Raspberries used is the Raspberry Pi 4 Model B. Below are the most relevant specifications of the component and an image of the device.

| **Specifications** | |
|---|---|
| **Processor** | Broadcom BCM2711, Quad core Cortex-A72 (ARM v8) 64-bit SoC @ 1.5GHz |
| **RAM** | 2GB LPDDR4-3200 SDRAM |
| **Network** | 2.4 GHz and 5.0 GHz IEEE 802.11ac wireless, Bluetooth 5.0, BLE<br>Gigabit Ethernet |
| **Hard drive** | 16GB micro-SD card for loading operating system and data storage |
| **Screen ports** | 2 × micro-HDMI ports (up to 4kp60 supported) |

Table 2. Raspberry Pi 4 Model B specifications

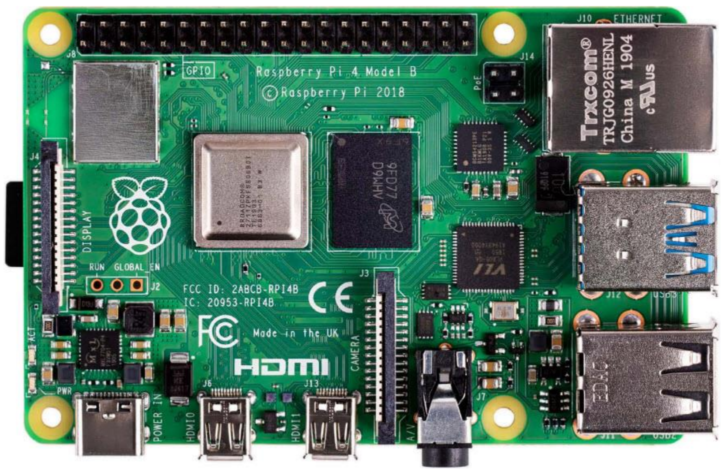


Figure 3. Raspberry Pi 4 Model B picture

One of the peculiarities of the system is its ARM processor, which is one of the group of CPUs based on the RISC (Reduced Instruction Set Computer) architecture developed by ARM (Advanced RISC Machines).

Although it was first used in the 1980s, it is still widely used until this day in millions of machines due to its several advantages. The fact that the number of instructions that ARM processors need to understand is very limited, makes the design of the processor simpler which means reduced complexity in its circuits, less transistors, smaller size, less power consumption. On the other hand, they are less powerful than other processor architectures used in desktop PCs.

The processor architecture will play an important role in this study as will be observed later.

Likewise, the fact that the hard drive is a micro-SD card with little capacity, the specification is 16GB, will also be a limitation in different stages of the process that will impact some of the procedures (e.g., build the Docker image) and to some extent the performance of the computation.

## 2.3. Network configuration

Some configuration is essential so that both nodes can communicate with each other via the local area network.

First, in both nodes disable the DHCP[5] (Dynamic Host Configuration Protocol) configuration so a static IP[6] (Internet Protocol) address can be assigned to them. This gives us the ability to connect to the node with SSH[7] (Secure Shell) knowing the IP address beforehand.

[5] Link to https://en.wikipedia.org/wiki/Dynamic_Host_Configuration_Protocol
[6] Link to https://en.wikipedia.org/wiki/Internet_Protocol
[7] Link to https://en.wikipedia.org/wiki/Secure_Shell

This is achieved by editing the */etc/dhcpcd.conf* file in every node with the configuration described below.

```
interface eth0
static ip_address=192.168.0.{x}/24
static routers=192.168.0.10
static domain_name_servers=192.168.0.10
```

Where the *eth0* is the first ethernet interface, the *ip_address* is the exact IP address that should be assigned to each node after replacing the *{x}* variable, the *routers* tell the nodes how to connect to the internet and the *domain_name_servers* are the DNS. Therefore, the IP address *192.168.0.10* belongs to the router.

The following table shows the configuration used for node0 (i.e., the front-end) and the node1 (i.e., the worker).

| Node | Topology | IP address |
|---|---|---|
| **node0** | Front-end | 192.168.0.1/24 |
| **node1** | Worker | 192.168.0.2/24 |

Table 3. IP addresses for the front-end and the worker nodes

Afterwards, edit the file */etc/hosts,* used by the DNS to identify the IP addresses with a name, with the following details with the aim to facilitate the reference to the worker node from the front-end node.

```
192.168.0.1 node0
192.168.0.2 node1
```

Finally, from the node0 install the SSH key to the node1. The goal is to provision access without requiring a password for each login. This facilitates the work from the node0 that will need to communicate with node1 to schedule some work.

Indeed, Open MPI requires that jobs can be started on remote nodes without any input from the keyboard. For example, if using rsh or ssh as the remote agent, you must have your environment setup to allow execution on remote nodes without entering a password or passphrase.

To generate the SSH key from the node0 it is used the *ssh-keygen* command, which creates a key par (public and private keys), and then with the command *ssh-copy-id node1* it is installed as an authorized key on the worker node.

## 2.4. Operating system

The operating system installed in the Raspberries is a Linux distribution based on Debian[8].

```
~                                                                 x
pi@node0:~ $ cat /etc/os-release
   PRETTY_NAME="Raspbian GNU/Linux 11 (bullseye)"
   NAME="Raspbian GNU/Linux"
   VERSION_ID="11"
   VERSION="11 (bullseye)"
   VERSION_CODENAME=bullseye
   ID=raspbian
   ID_LIKE=debian
   HOME_URL="http://www.raspbian.org/"
   SUPPORT_URL="http://www.raspbian.org/RaspbianForums"
```

Albeit the Raspberries support ARM v8, as described above in the specifications, they are running in ARM v7 mode with a 32 bits operating system. This is an important detail to bear in mind while working on the compilation of the libraries and building the Docker image.

Note that the installation of the operating system and the basic set up, aside from the network configuration, is out of the scope of the study. For reference, only the following pitfall is going to be described.

As a side note, one of the pitfalls that was identified in the default SO configuration was that when running MPI from node0 and assigning some work to node1, the worker node couldn’t successfully execute the operation because the environment variables were missing. However, running SSH into the node itself and running the command explicitly from within it, the environment variables were loaded. Further investigations identified that the *~/.bashrc* file had a snippet to scape if not running interactively.

```
# If not running interactively, don't do anything
case $- in
   *i*) ;;
     *) return;;
esac
```

It is strongly suggested to comment out those lines in the source file.

[8] Link to https://www.debian.org/

# 3. VIRTUALIZATION

## 3.1. Virtual machines

A system virtual machine[9] (VM) is a compute resource managed by a hypervisor that uses software instead of a physical computer to virtualize the entire machine, including the hardware, in order to make it behave as if it was a separate system. Even though virtual machines can run in parallel in the same host and under the hood they all share the same hardware, each virtual machine runs its own operating system and functions separately from the others.

The abstraction they bring is powerful as they give us the ability to run applications that have completely different requirements than the host, such as a different operating system, make changes or install programs without incurring in the risk of affecting negatively the host, for instance in the event of installing a virus or malware, or even test applications that must be isolated from anything else.

As we can observe, the advantages of virtual machines are numerous, however, they come at a cost. They are a very high resource computing software that uses very heavily the capabilities of the physical system, running more than one virtual machine on the same host can result in a noticeable performance degradation that can make things unstable, and booting them or tearing them down is very slow. Additionally, they usually require a considerable amount of space in the hard drive to work.

Apart from the issues with the resources, setting up a virtual machine is a time-consuming process that requires some knowledge in computing systems, as you will need configure a set of parameters that define the physical resources that will be allocated to the VM at runtime, you will need to find the image of the operating system you want to install in it, go through all the installation process and finally configure the SO post installation.

Like virtual machines, it appeared the containers, a different technology that is a lightweight version of the VMs that have a different target and intend to cover the problem from a different point of view.

## 3.2. Containers

The most noticeable difference between the two solutions is that only virtualize the software layers on top of the operating system, as opposite to VMs that virtualize the entire computer including the hardware.

[9] Link to https://en.wikipedia.org/wiki/Virtual_machine

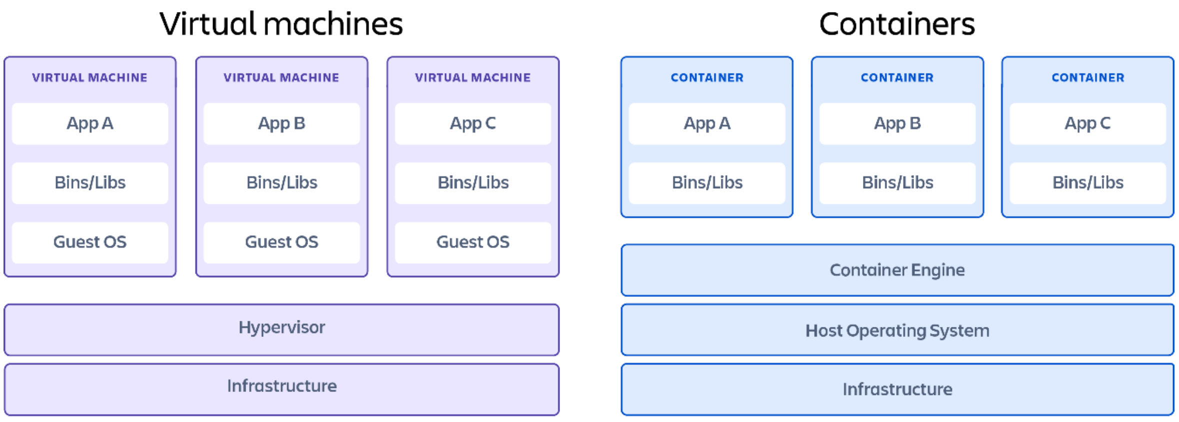


Figure 4. Comparison between virtual machines and containers

Because containers run on top of the OS, they are a lightweight software package that only needs to contain all the dependencies and libraries required to execute the applications.

It is worth noting that containers can and will run its own version of the OS that might be different than the one running in the host. However, unlike VMs, the OS is already prepared and configured to run on the fly with no extra effort, so users can focus on including the necessary dependencies like system libraries, external third-party code packages, etc.

The level of isolation of containers is also lower, meaning that all of them share the same underlying hardware and operating system layer on the host, on one hand, this increases the risk of vulnerabilities reaching the host or the other containers, on the other hand, with some network configuration you can make the containers communicate with each other.

Probably one of the most popular and widely used container providers is Docker, used in this study in combination with Singularity.

# 4. LIBRARIES OVERVIEW

## 4.1. OpenMPI

*Version used of the library is v4.1.3.*

OpenMPI[10] is the chosen implementation of the MPI standard, the Message Passing Interface-programming paradigm used to perform the parallel processing in the simulation of the HPC application

Open MPI is an open-source Message Passing Interface implementation that is developed and maintained by a consortium of academic, research, and industry partners. Open MPI is therefore able to combine the expertise, technologies, and resources from all across the High-Performance Computing community in order to build the best MPI library available. Open MPI offers advantages for system and software vendors, application developers and computer science researchers.

Although there are other very competent alternatives in the market, such as MPICH, the reason why OpenMPI was chosen over the rest is because OpenMPI seems to have a higher adoption and thus it is easier to find information and curated documentation on the internet.

## 4.2. OpenFOAM

*Version used of the library is v2112.*

OpenFOAM[11] is the distributed scientific application developed by OpenCDF Ltd that leverages the MPI standard and that will be used in this study.

OpenFOAM stands for *Open-source Field Operation And Manipulation* and is a C++ toolbox for the development of customised numerical solvers and post-processing utilities for the solution of computational fluid dynamics[12] (CFD), that over the recent years has emerged as an important approach in chemical and biochemical engineering.

Nevertheless, the distributed application is not very relevant as long as it supports parallel processing with the MPI implementation selected.

[10] Link to https://www.open-mpi.org/
[11] Link to https://www.openfoam.com/
[12] Link to https://www.sciencedirect.com/topics/engineering/computational-fluid-dynamic

## 4.3. Extrae

*Version used of the library is v4.0.0.*

Extrae[13], developed by the Performance Tools group[14] at the Barcelona Supercomputing Center, is a dynamic instrumentation package to trace programs compiled and run with the shared memory, the message passing (MPI) programming model or both programming models. Its main function is to generate trace-files of the execution of a distributed application that can be later visualized with Paraver.

Each line of these trace-files represents an event which occurred during the execution of the code. There are different types of events such as CPU activity, communication or user events.

In this study, Extrae will be used to trace the distributed execution of OpenFOAM using the MPI programming paradigm. The shared memory model is not targeted in this study.

## 4.4. Paraver

*Version used of the library is v4.10.0.*

Paraver[15], that stands for PARAllel Visualization and Events Representation, is the trace visualisation and analysis browser that leverages the trace-files generated by Extrae to evaluate the performance obtained while running a single application in a distributed mode.

PARAVER is based on a simple interface to manage several displaying windows that provides many functionalities to see and analyse quantitatively the trace file.

[13] Link to https://tools.bsc.es/extrae
[14] Link to https://www.bsc.es/discover-bsc/organisation/scientific-structure/performance-tools
[15] Link to https://tools.bsc.es/paraver

# 5. LIBRARIES COMPILATION

## 5.1. Context

In order to build the Docker image, first it is necessary to pre-compile for the current architecture (ARM v7 32 bits) all the necessary libraries to carry out the study.

The compilation has been carried out in a single node, the front-end, due to different reasons:

- Ensure that the binaries result of the compilation are compatible with the processor architecture of the Raspberries.
- Get familiarised with the system.
- Run the tools in an environment that faithfully reflects the end system.

To start with the compilation, it is necessary a host connected to the same local area network (e.g., a laptop or desktop machine) that connects to node0 via SSH.

```
albert:~ $ ssh pi@192.168.0.1
pi@192.168.0.1's password:
…
pi@node0:~ $
```

After accessing the shell in node0, the different libraries can be compiled as will be described in the following sections.

Before going into detail, it is worth mentioning that the hard drive capacity in the Raspberries is very limited and since some of the libraries require a considerable amount of space after compiling, it is not possible to build the Docker image in the node due to space constraints.

For example, only OpenFOAM requires ~2.5GB after compile. This means that building a Docker image that just contains OpenFOAM would require ~7.5GB of space available because the build context is copied over to the Docker daemon before the build begins.

Not only that, but also the read/write speed of the micro-SD is low which makes the build process even more difficult and time consuming.

The workaround to this problem is to copy the compiled libraries from node0 to an external host and build the Docker image there. This can be done with the secure-copy command-line that relies on SSH and that allows you to securely copy files and directories between two locations.

For instance, let’s assume that the node0 has OpenMPI compiled in the */opt* directory and that the whole folder should be copied to the host in the */home* directory, the command-line would look like this.

```
~                                                                          x
albert:~ $ scp pi@192.168.0.1:/opt/openmpi-4.1.3 /home
```

Finally, it is noteworthy that building the Docker image outside of the HPC cluster is not a problem, as ultimately the image will be pushed to the Docker Registry Hub[16] and will be available from anywhere that has access to the internet.

## 5.2. Compile OpenMPI

The steps to compile OpenMPI version 4.1.3 from inside node0 are relatively straight forward, except the configuration step prior to the compilation, which requires some arguments to ensure that the compilation can be successfully ported to other machines.

Prior to download the source code of OpenMPI from the official website, it is necessary to download the GNU Wget package for retrieving files using HTTP and HTTPS. Super user privileges might be required.

```
~                                                                          x
pi@node0:~ $ sudo apt-get update && apt-get install -y wget
pi@node0:~ $ wget https://download.open-mpi.org/release/open-
   mpi/v4.1/openmpi-4.1.3.tar.gz
```

Afterwards, extract the compressed file, remove it and run the *configure* script to set up the installation.

```
~                                                                          x
pi@node0:~ $ tar -xvzf openmpi-4.1.3.tar.gz
pi@node0:~ $ rm openmpi-4.1.3.tar.gz
pi@node0:~ $ cd openmpi-4.1.3
pi@node0:/openmpi-4.1.3 $ ./configure --prefix=/opt/openmpi-4.1.3 --enable-
   mpirun-prefix-by-default --disable-mca-dso --enable-static
```

[16] Link to https://hub.docker.com/

It is important to explain the arguments provided in the *configure* step.

| Configure arguments | |
|---|---|
| Prefix | Tells OpenMPI the install directory. |
| enable-mpirun-prefix-by-default | This will make mpirun behave exactly the same as "*mpirun --prefix $prefix ...*", where *$prefix* is the value given to *--prefix* in configure. |
| disable-mca-dso | Causes all plugins to be built as part of Open MPI's main libraries. However, does not affect whether OpenMPI's main libraries are built as static or shared. |
| enable-static | Causes the building of static libraries (e.g., libmpi.a). |

Table 4. Configure arguments for Open MPI library

Then, compile the source code using all the cores available in node0 and install the library in the directory defined in the *prefix* argument.

```
pi@node0:/openmpi-4.1.3 $ make -j 4
pi@node0:/openmpi-4.1.3 $ make install
```

Finally, add to the PATH variable the install directory of OpenMPI.

```
pi@node0:/openmpi-4.1.3 $ export PATH=/opt/openmpi-4.1.3/bin:$PATH
pi@node0:/openmpi-4.1.3 $ which mpirun
   /opt/openmpi-4.1.3/bin/mpirun
pi@node0:/openmpi-4.1.3 $
```

From the libraries used in this study, OpenMPI is the only library that needs to be configured in the front-end as well as in the container. The rationale is that the MPI process is initiated from the host as described in a later chapter.

## 5.3. Compile OpenFOAM

The configuration to install OpenFOAM is going to be based on OpenMPI implementation.

First, it is necessary to define some environment variables related to MPI.

```
pi@node0:~ $ export MPI_ROOT=/opt/openmpi-4.1.3
pi@node0:~ $ export MPI_ARCH_FLAGS="-DOMPI_SKIP_MPICXX"
pi@node0:~ $ export MPI_ARCH_INC="-isystem $ MPI_ROOT /include"
pi@node0:~ $ export MPI_ARCH_LIBS="-L$MPI_ROOT/lib -lmpi"
pi@node0:~ $ export LD_LIBRARY_PATH=$MPI_ROOT/lib
pi@node0:~ $ export PATH=$MPI_ROOT/bin:$PATH
```

Afterwards, download the source code, extract it and then remove the file.

```
pi@node0:~ $ wget https://dl.openfoam.com/source/v2112/OpenFOAM-v2112.tgz
pi@node0:~ $ tar -xvzf OpenFOAM-v2112.tgz
pi@node0:~ $ rm OpenFOAM-v2112.tgz
```

Before compiling the code, it’s important to tell OpenFOAM what MPI library to use and that it must be compiled with the optimal option. These settings can be defined in the ~/OpenFOAM-v2112/etc/bashrc.

```
export WM_MPLIB=SYSTEMOPENMPI
export WM_COMPILE_OPTION=Opt
```

Additionally, it is required to modify the way in which the compiler will create the floating-point instructions, to make it compatible with the processor architecture. This can be achieved by updating the *-mfloat_abi* parameter from *softfp* to *hard* in the file *~/OpenFOAM-v2112/wmake/rules/linuxARM7Gcc/cOpt* and in the file *~/OpenFOAM-v2112/wmake/rules/linuxARM7Gcc/c++Opt*.

After setting everything up, we can proceed to compile the code. If the compilation is successful, the last step is to verify that the environment variables defined in OpenFOAM can be successfully exported.

```
pi@node0:~ $ cd OpenFOAM-v2112
pi@node0:/OpenFOAM-v2112 $ source etc/bashrc
   No completions for /opt/OpenFOAM-v2112/platforms/
   linuxARM7GccDPInt32Debug/bin
   [ignore if OpenFOAM is not yet compiled]
pi@node0:/OpenFOAM-v2112 $ ./Allwmake -j 1
```

```
   ==========================================
   Starting compile OpenFOAM-v2112 Allwmake
   Gcc system compiler
   linuxARM7GccDPInt32Debug, with SYSTEMOPENMPI sys-openmpi
   ==========================================
   ...
pi@node0:/OpenFOAM-v2112 $ cd ..
pi@node0:~ $ source OpenFOAM-v2112/etc/bashrc
pi@node0:~ $
```

The compilation can take several hours, on my experience it took about ~14 hours. Indeed, this was a problem in the beginning, since we were connecting to node0 remotely and it was causing a TTY timeout after a few minutes that was cancelling the compilation process.

The workaround for that issue was to use tmux[17], a terminal multiplexer. It lets you switch easily between several programs in one terminal, detach them (they keep running in the background) and reattach them to a different terminal.

### 5.4. Compile Extrae

Similarly, to the previous libraries, the first step is to download the source code from the official Barcelona Supercomputing Center tools website[18].

```
~                                                                    x
pi@node0:~ $ wget https://ftp.tools.bsc.es/extrae/extrae-4.0.0-src.tar.bz2
pi@node0:~ $ apt-get install bzip2
pi@node0:~ $ tar -xvf extrae-4.0.0-src.tar.bz2
pi@node0:~ $ rm extrae-4.0.0-src.tar.bz2
```

It's worth noting that in order to be able to decompress the file, compressed with bzip2, it is necessary to install the corresponding library. Additionally, during the process, it was identified some other indispensable dependencies that required to be installed to successfully compile Extrae.

```
~                                                                    x
pi@node0:~ $ apt-get install gfortran
pi@node0:~ $ apt-get install build-essential
```

[17] Link to https://github.com/tmux/tmux/wiki
[18] Link to https://tools.bsc.es/

```
pi@node0:~ $ apt-get install libiberty-dev
pi@node0:~ $ apt-get install binutils-dev
pi@node0:~ $ apt-get install libxml2-dev
```

Indeed, some of these libraries will also need to be explicitly installed within the container since they are required at runtime by Extrae and are not included in the base image.

Next is to issue the configuration command.

```
~                                                                                      x
pi@node0:~ $ cd extrae-4.0.0
pi@node0:/extrae-4.0.0 $ ./configure --prefix=/opt/extrae-4.0.0 --without-unwind
    --without-dyninst --without-papi --with-mpi=/opt/openmpi-4.1.3
    --enable-posix-clock --with-binutils=/usr
```

The following table contains a breakdown of the arguments provided in the *configure* step.

| **Configure arguments** | |
|---|---|
| Prefix | Tells Extrae the install directory. |
| without-unwind | Specifies that the Unwind[19] libraries should not be used. These libraries are used to get call stack information on several architectures. |
| without-dyninst | Specifies that the Dyninst[20] package should not be used. DynInst is a third-party instrumentation library that gives the flexibility to add instrumentation to the application without modifying the source code. |
| without-papi | Specifies that the PAPI[21] libraries should not be used. PAPI stands for Performance Application Programming Interface and provides a consistent interface and methodology for use of the performance counter hardware found in most of the microprocessors. |

[19] Link to https://www.nongnu.org/libunwind/
[20] Link to https://www.dyninst.org/
[21] Link to https://icl.utk.edu/papi/

| | |
|---|---|
| enable-posix-clock | Use POSIX clock (*clock_gettime call*) instead of low-level timing routines. It is recommended to use this option if the system where you install the instrumentation package modifies the frequency of its processors at runtime. |
| binutils | Specifies the location for the binutils package. The binutils package is necessary to translate addresses into source code references. |

Table 5. Configure arguments for Extrae library

Finally, run the build and installation commands.

```
~                                                          x
pi@node0:~ $ make
pi@node0:~ $ make install
```

Extrae uses an interposition mechanism done by the runtime loader that substitutes the original symbols of the binaries by those provided by the instrumentation package. It leverages the Linux dynamic pre-loader (i.e., LD_PRELOAD) environment variable, which contains one or more paths to shared libraries or shared objects that the Linux loader will load before any other shared library.

By default, Extrae tries to preload Fortran version of the *libmpitrace* library to instrument MPI calls for apps in that language. Since OpenFOAM is implemented in C, it is necessary to update the following script to make the package point to the correct library. Additionally, we will take the opportunity to update the path for the *EXTRAE_HOME* and *EXTRAE_CONFIG_FILE* environment variables to point them to the install directory defined in the previous step.

This can be done by updating the content of the *trace.sh* script, located in */opt/extrae-4.0.0/share/example/MPI/ld-preload*, as follows.

```
#!/bin/sh


export EXTRAE_HOME=/opt/extrae-4.0.0/etc/extrae.sh
export EXTRAE_CONFIG_FILE=${EXTRAE_HOME}/share/example/MPI/
        extrae.xml
```

```
export LD_PRELOAD=${EXTRAE_HOME}/lib/libmpitrace.so


## Run the desired program
$*
```

The environment variable that references the XML configuration file is one of the most important settings, as it tells Extrae what are the traces that must generate on runtime.

## 5.5. Compile Paraver

Visualising the trace files generated by Extrae is a long and thorough process that must be done after executing OpenFOAM, and after generating and merging the intermediate trace files. It is an asynchronous procedure, independent to the execution of the application, which requires the intervention of a human.

Certainly, if we think about the benefits and the purpose of containers, we can realise that including Paraver within the container doesn't fit well.

Probably, the most rational thing is to have Paraver prepared on the front-end or, more likely, to export the trace files to an external host other than the supercomputer and perform the analysis there.

It is due to this fact that compiling Paraver doesn't make sense and it is suggested to use one of the already compiled binaries that they offer in their website[22]. The executables are available for the most common architectures, including Linux, MacOS and Windows.

In this study, the approach followed will be to extract the trace files to an external host using the secure-copy command-line.

[22] Link to https://ftp.tools.bsc.es/wxparaver/

# 6. DOCKER

## 6.1. Dockerfile

Docker can build images automatically by reading the instructions from a *Dockerfile*. A *Dockerfile* is a text document that contains all the commands a user could call on the command line to assemble an image. Using the *docker build* command we can create an automated build that executes several command-line instructions in succession.

The following code describes the Dockerfile used to generate the Docker image used in all the scenarios for this study.

```
FROM arm32v7/debian:11.3-slim

# Install SSH, VIM editor and dependencies needed not included in the
# base image
RUN apt-get update && apt-get install -y \
  ssh \
  build-essential \
  libiberty-dev \
  binutils-dev \
  libxml2-dev \
  vim

# Set environment variables
RUN echo "export LD_LIBRARY_PATH='/opt/openmpi-
    4.1.3/lib':$LD_LIBRARY_PATH" >> ~/.bashrc
RUN echo "export PATH='/opt/openmpi-4.1.3/bin':$PATH" >> ~/.bashrc
RUN echo "export OMPI_ALLOW_RUN_AS_ROOT=1" >> ~/.bashrc
RUN echo "export OMPI_ALLOW_RUN_AS_ROOT_CONFIRM=1" >> ~/.bashrc
RUN echo "export OMPI_MCA_btl_vader_single_copy_mechanism=none" >>
    ~/.bashrc
RUN echo "source /opt/OpenFOAM-v2112/etc/bashrc" >> ~/.bashrc

# Copy pre-compiled libraries from host
COPY /opt/OpenFOAM-v2112 /opt/OpenFOAM-v2112
COPY /opt/openmpi-4.1.3 /opt/openmpi-4.1.3
COPY /opt/extrae-4.0.0 /opt/extrae-4.0.0
```

```
# Add all permissions to OpenFOAM directory so mpirun has write privileges
RUN chmod -R 777 /opt/OpenFOAM-v2112

ENTRYPOINT ["/bin/bash"]
```

Even though the Dockerfile is self-descriptive, there are some instructions that are not so obvious. Below is described the most relevant sets of operations that it contains.

```
FROM arm32v7/debian:11.3-slim
```

The very first line describes the base image used to build our custom one on top of it. The base image is based on the lightweight version of Debian 11.3 for ARM processors using 32 bits. It is very important that the base image is compatible with the architecture that the processors of the Raspberries are built in.

```
RUN apt-get update && apt-get install -y \
…
```

It updates the packages list to fetch the most recent available information in the repositories. The following lines install some of the libraries that are identified as missing to run the tools.

```
RUN echo "export LD_LIBRARY_PATH='/opt/openmpi-
   4.1.3/lib':$LD_LIBRARY_PATH" >> ~/.bashrc
RUN echo "export PATH='/opt/openmpi-4.1.3/bin':$PATH" >> ~/.bashrc
RUN echo "export OMPI_ALLOW_RUN_AS_ROOT=1" >> ~/.bashrc
RUN echo "export OMPI_ALLOW_RUN_AS_ROOT_CONFIRM=1" >> ~/.bashrc
```

The first two instructions are the environment variables necessary to run OpenMPI. The next two instructions explicitly tell OpenMPI to run even with as root, which is the default user within the container in Docker.

```
RUN echo "export OMPI_MCA_btl_vader_single_copy_mechanism=none" >>
   ~/.bashrc
```

This instruction disables the cross-memory-attach (CMA). OpenMPI has many ways to transfer messages between ranks. If the ranks are on the same node, it is faster to do these transfers using shared memory rather than involving the network stack. One of the approaches is to use system calls to transfer messages directly from Rank A’s virtual memory to Rank B’s, which is known CMA, and it gives significant performance improvements in benchmarks. However, in this case it cases some errors and turning off the single copy addressed them.

```
RUN echo "source /opt/OpenFOAM-v2112/etc/bashrc" >> ~/.bashrc
```

Loads the configuration for OpenFOAM to run and to load in the current session the path to the binaries and the libraries.

```
COPY /opt/OpenFOAM-v2112 /opt/OpenFOAM-v2112
COPY /opt/openmpi-4.1.3 /opt/openmpi-4.1.3
COPY /opt/extrae-4.0.0 /opt/extrae-4.0.0
```

This set of instructions copies the tools that were compiled in the front-end to the image, so that the container has them available at runtime.

```
RUN chmod -R 777 /opt/OpenFOAM-v2112
```

Running the option *simpleFoam* from OpenFOAM, which has the responsibility of executing the simulation, requires write permissions on the directory as it needs to generate some directories and files. It was observed that combining this with the command *mpirun* to parallelize the simulation resulted in some errors due to missing privileges.

```
ENTRYPOINT ["/bin/bash"]
```

The last instruction tells Docker to use *bash* by default when running an executable or when connecting interactively inside the container.

## 6.2. Build image

As mentioned in previous chapters, building the image will take place outside of the front-end due to space constraints.

First, it is necessary to copy compiled tools to the external computer.

```
~                                                                 x
albert:~ $ scp pi@192.168.0.1:/opt/openmpi-4.1.3 /home
albert:~ $ scp pi@192.168.0.1:/opt/OpenFOAM-v2112 /home
albert:~ $ scp pi@192.168.0.1:/opt/extrae-4.0.0 /home
```

Afterwards, move to the directory where the tools were copied to and build the docker image. Note the *-t* argument used to tag the image.

```
albert:~ $ cd home
albert:home $ docker build -t asanuy/openmpi:1.0.0 .
```

## 6.3. Publish image

In order to make accessible the Docker image from the front-end node, it is possible to either copy the image from the external computer to the Raspberries or push the image to the public Docker hub[23].

The suggested approach is to publish it to the public repository. First, it is necessary to create a new account or sign in to https://hub.docker.com.

Afterwards, from the command-line, log in to the Docker Hub account.

```
albert:~ $ docker login
```

After successfully login in with the correct credentials, publish the image to the registry with the *push* command.

```
albert:~ $ docker push asanuy/openmpi:1.0.0
```

In the event of downloading the image from the public registry, it can be done with the *pull* command.

```
albert:~ $ docker pull asanuy/openmpi:1.0.0
```

[23] Link to https://hub.docker.com

# 7. SINGULARITY

## 7.1. Installation

Before start using Singularity, it must be installed the development libraries to both the front-end and the worker.

```
~                                                                x
pi@node0:~ $ sudo apt-get update && sudo apt-get install -y \
    build-essential \
    libssl-dev \
    uuid-dev \
    libgpgme11-dev \
    squashfs-tools
```

Since Singularity 3.0 is written primarily in Go, it is needed to download the package, install it and configure it.

```
~                                                                x
pi@node0:~ $ wget https://dl.google.com/go/go1.18.3.darwin-arm64.tar.gz
```

Extract the archive to */SHARED*, so both the front-end and the worker have access to the directory.

```
~                                                                x
pi@node0:~ $ sudo tar -C /SHARED -xzf wget go1.18.3.darwin-arm64.tar.gz
```

Then, set up your environment for Go.

```
~                                                                x
pi@node0:~ $ echo 'export GOPATH=/SHARED/go' >> ~/.bashrc
pi@node0:~ $ echo 'export PATH/SHARED/go/bin:${PATH}:${GOPATH}/bin' >>
    ~/.bashrc
pi@node0:~ $ source ~/.bashrc
pi@node0:~ $ rm go1.18.3.darwin-arm64.tar.gz
```

Next, clone the singularity repository.

```
~                                                                x
pi@node0:/SHARED $ cd /SHARED
pi@node0:/SHARED $ mkdir -p $GOPATH/src/github.com/sylabs
```

```
pi@node0:/SHARED $ cd $GOPATH/src/github.com/sylabs
pi@node0:/SHARED/go/src/github.com/sylabs $ git clone
   https://github.com/sylabs/singularity.git
pi@node0:/SHARED/go/src/github.com/sylabs $ cd singularity
```

Finally, compile the Singularity binary and again place it in the */SHARED* directory so it is accessible by all the nodes.

```
~                                                                                x
pi@node0:/SHARED/go/src/github.com/sylabs/singularity $ ./mconfig
pi@node0:/SHARED/go/src/github.com/sylabs/singularity $ make -C
   /SHARED/singularity
pi@node0:/SHARED/go/src/github.com/sylabs/singularity $ sudo make -C builddir
   install
```

At this point, singularity should be available both to the front-end and the worker node.

## 7.2. Singularity Image File

First, it’s necessary to create the Singularity Image File (i.e., SIF), similar to what was done with Docker. However, since we are going to use the SIF as a wrapper of the Docker image with minimum configuration, the configuration is straightforward because we are taking advantage of all the previous work.

```
Bootstrap: docker
From: asanuy/openmpi:1.0.0

%environment
  export LD_LIBRARY_PATH='/opt/openmpi-4.1.3/lib':$LD_LIBRARY_PATH
  export PATH='/opt/openmpi-4.1.3/bin':$PATH
  export OMPI_MCA_btl_vader_single_copy_mechanism=none
```

Those very few lines in the definition file, are enough to build  a Singularity image with the *build* command, where *openmpi.sif* is the Singularity image file and *openmpi.img* is the resulting image built.

```
~                                                                                x
pi@node0:~ $ cd /SHARED
pi@node0:SHARED $ sudo singularity build openmpi.img openmpi.sif
```

Like Docker, Singularity needs to reference an image in order to run a container.

# 8. RUNNING THE CONTAINER

## 8.1. Case study

In order to better understand what running the container means, it is essential to briefly explain which exactly is the simulation that will be run and how OpenFOAM works.

The OpenFOAM library comes with a collection of examples to run different simulations, where some of them are more complex and expensive than others. For the study, it was chosen one case from the examples called *pitzDaily* that is thought to investigate steady turbulent flow over a backward-facing step, although the details of the simulation are negligible and anecdotic to this study, it is important to choose one case that is feasible to process and complete by the Raspberries in an acceptable amount of time. Additionally, it is crucial to always stick to the same case so the results can be compared objectively.

Every case in OpenFOAM is designed to be executed from a terminal command-line, typically reading and writing a set of data files associated with a particular case, where the data files for a case are stored in a directory named after the case.

Running a case always starts with the *blockMesh* command. The principle behind *blockMesh* is to decompose the domain geometry into a set of 1 or more three dimensional, hexahedral blocks.

Afterwards, the mesh and fields must be decomposed using the *decomposePar* utility to run OpenFOAM in parallel on distributed processors using MPI. The *decomposePar* command breaks up the domain with minimal effort but in such a way to guarantee an economic solution. The geometry and fields are broken up according to a set of parameters specified in a dictionary named *decomposeParDict* that must be located in the system directory of the case of interest.

For this study, it will be used the following dictionary, where the number of subdomains will vary depending on the number of MPI processes. Note that the number of subdomains must match the coefficient *XYZ*. For instance, this would be the decomposition if it was used 4 MPI processes.

```
FoamFile
{
    version     2.0;
    format      ascii;
    class       dictionary;
    object      decomposeParDict;
}

numberOfSubdomains 4;

method          simple;

coeffs
{
    n          (2 1 2);
}
```

Finally, the simulation can be run in parallel with the command *simpleFoam -parallel*; note the parallel flag which tells OpenFOAM to run in distributed mode by using all the processors by using MPI.

## 8.2. Execution

After the long process of compiling and configuring numerous resources, everything is set up to finally run the Singularity container from the front-end node.

Since running the application has some complexity due to how OpenFOAM works, it has been created the following bash script which contains all the commands in order to facilitate the procedure and make it less prone to error.

```
#!/bin/bash

echo Loading OpenFOAM environment variables
source /opt/OpenFOAM-v2112/etc/bashrc

export SIMULATION_DIR=/opt/OpenFOAM-
v2112/tutorials/incompressible/simpleFoam/pitzDaily

echo Running blockMesh
blockMesh -case $SIMULATION_DIR

echo Creating decomposePar dictionary in the simulation directory
cp decomposeParDict $SIMULATION_DIR/system

echo Running decomposePar
```

```
decomposePar -case $SIMULATION_DIR

echo Loading Extrae environment variables
source /opt/extrae-4.0.0/share/example/MPI/ld-preload/trace.sh

echo Running simpleFoam in parallel
simpleFoam -parallel -case $SIMULATION_DIR
```

The script, with name *run.sh*, assumes that the *decomposeParDict* file is located in the same current directory. Besides that, it requires execution permissions.

```
~                                                                 x
pi@node0:/SHARED $ sudo chmod u+x run.sh
```

All the instructions in the script are related to OpenFOAM except the part where Extrae is loaded. That is precisely the command that enables the Extrae library in order to start instrumenting the runtime application and generating the traces.

Since it uses an interposition mechanism that is done by the runtime loader by substituting the original symbols by those provided by the instrumentation package, it is very important that Extrae is enabled at that exact point, right before starting the simulation, otherwise it will try to instrument all the commands and cause unexpected errors.

Finally, run the container via MPI with one single command. When running code within a Singularity container, it should not be referenced the MPI executables located inside it (e.g., *singularity exec mpirun -n 4 simpleFoam -parallel*). Instead, it should be used the MPI installation on the front-end to run Singularity and start an instance of the application from within a container for each MPI process.

Behind the scenes, OpenFOAM is linking to the MPI libraries from the MPI install within our container and these are, in turn, communicating with the MPI daemon on the host system.

```
~                                                                 x
pi@node0:/SHARED $ mpirun -n 4 -host node0:2,node1:2 -quiet --mca btl_tcp_if_include
192.168.0.1/24,192.168.0.2/24 singularity exec --writable-tmpfs /SHARED/openmpi.img
./run.sh
```

| Command arguments | |
|---|---|
| mpirun | Executes serial and parallel jobs in Open MPI. |
| -n <num> | Launch num processes per node on all allocated nodes. |
| -hosts <host1,host2,…hostN> | List of hosts on which to invoke processes. Note that the host's name can contain a colon followed by a number, that indicates how many processes must execute a given host. |
| -quiet | Suppress informative messages from orterun during application execution. |
| --mca btl_tcp_if_include <ip1,ip2,…ipN> | It is a network configuration over TCP that defines exactly what the addresses of the hosts are.<br>Not providing the addresses explicitly proved to give some errors due to MPI doing look ups on all the network interfaces available. |
| singularity exec | Tells Singularity to run a command within a container. |
| --writable-tmpfs | Makes the file system accessible as read-write with non-persistent data.<br>This is fundamental so OpenFOAM can write the data of the runtime simulation. |
| /SHARED/openmpi.img | The Singularity image. |
| ./run.sh | The custom script that acts as a wrapper to a set of instructions to facilitate the process. |

Table 6. Command arguments to run the container

Note that the Singularity container must be executed from the /SHARED directory, because it is the directory that all the nodes have read/write access to.

The rationale is that by default Extrae will try to produce the trace files in the current directory inside the container, but since Singularity by default mounts the current directory on the host system to the directories within the container, giving them the ability to write to the host itself, it will cause that all the containers will produce the traces in the shared folder of the front-end system.

This makes possible that the merger process, which takes place at the end once the application run has completed, can access all the intermediate traces and generate three final files, which will be permanent and accessible from the front-end even after the Singularity containers have been successfully completed and shut down.

One of the three files generated is the Paraver trace itself (.prv file), that contains the records with a timestamp that represent the information gathered during the execution of OpenFOAM. The other file generated is the Paraver configuration file (.prc file), that contains a dictionary to translate the values contained in the Paraver trace into a human readable string. The last file generated (.row file), contains the distribution of the application across the cluster computation resources.

## 8.3. Extrae configuration file

In the previous custom script, it can be observed that Extrae is loaded with simply one line.

```
echo Loading Extrae environment variables
source /opt/extrae-4.0.0/share/example/MPI/ld-preload/trace.sh
```

The content of the *trace.sh* file was described previously (chapter *5.4 Compile Extrae)*, and although it was mentioned that it contains a reference to the XML configuration file that defines what exactly should be instrumented, we didn’t take the opportunity to explain what is included in it.

The XML contains crucial information such as the basic trace behaviour, the intermediate files that are meant to be generated (e.g., Paraver or Dimeas), the configuration to analyze different resources (e.g., MPI, OpenMP, Network, CUDA, OpenCL, Dynamic memory, etc), and the settings of the merge process.

Following is the XML configuration file used in the execution of the container.

```
<?xml version='1.0'?>

<!-- Enables the tracing and defines the tracing mode (detail/bursts), the dome dir and
the resulting traces -->
<trace enabled="yes"
 home="/opt/extrae-4.0.0"
 initial-mode="detail"
 type="paraver"
>
```

```
<!-- Configuration of some MPI dependant values -->
<mpi enabled="yes">
  <!-- Gather counters in the MPI routines -->
  <counters enabled="yes" />
  <!-- Capture all MPI_Comm_* calls -->
  <comm-calls enabled="yes" />
</mpi>

<!-- Emit information of the callstack -->
<callers enabled="yes">
  <!-- At MPI calls, select depth level -->
  <mpi enabled="yes">1-3</mpi>
  <!-- At sampling points, select depth level -->
  <sampling enabled="yes">1-5</sampling>
</callers>

<!-- Configure which software/hardware counters must be collected -->
<counters enabled="yes">
  <!-- Obtain resource usage information -->
  <resource-usage enabled="yes" />
</counters>

<!-- Buffer configuration -->
<buffer enabled="yes">
  <!-- How many events can we handle before any flush -->
  <size enabled="yes">5000000</size>
</buffer>

<!-- Do merge the intermediate tracefiles into the final tracefile
     Named according to the binary name
     options:
     synchronization = { default, task, node, no } (default is node)
     max-memory = Number (in Mbytes) max memory used in merge step
     joint-states = { yes, no } generate joint states
     keep-mpits = { yes, no } keep mpit files after merge
  -->
  <merge enabled="yes"
    synchronization="default"
    tree-fan-out="16"
    max-memory="512"
    joint-states="yes"
    keep-mpits="yes"
    translate-addresses="yes"
    sort-addresses="yes"
    translate-data-addresses="yes"
    overwrite="yes"
  />

</trace>
```

## 8.4. Analysis

The analysis will be focused on sharing the evidence collected during the execution of the application with different configurations, on the comparison of the Paraver trace files generated for each of them, and on trying to understand what was the behaviuor of the cluster.

From the multiple combinations of workloads available, with different processes and different hosts intervening in the execution, it was narrowed down to the following three scenarios.

Scenario 1. Two MPI processes on a single host, the front-end (node0).
Scenario 2. Four MPI processes on a single host, the front-end (node0).
Scenario 3. Four MPI processes on two hosts, the front-end (node0) and the worker (node1). The two hosts have the same number of processes.

Before proceeding to discuss the scenarios, there is an important observation to make. The Extrae library is going to play an important role in the process and for this reason it was considered important to describe its impact in each case.

**Scenario 1**

The execution of the OpenFOAM simulation in parallel took about 41 seconds in total time for the two cores that participated in the process. If we observe the view with the timeline of all the MPI calls, we can notice that there is no process taking more than the other and thus slowing everything down.

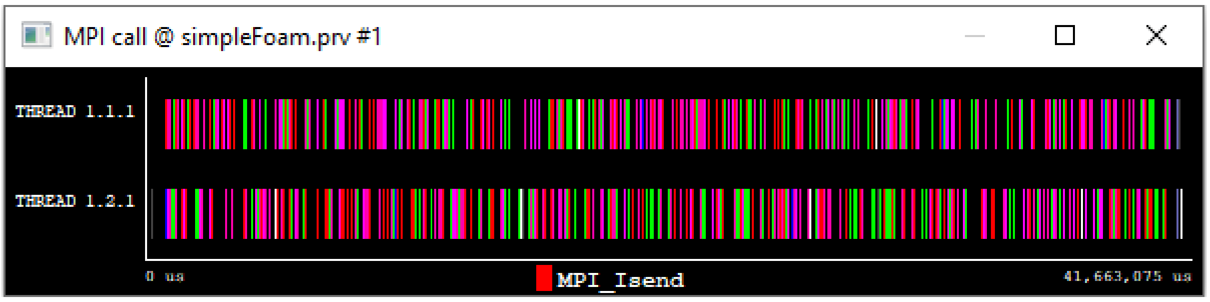
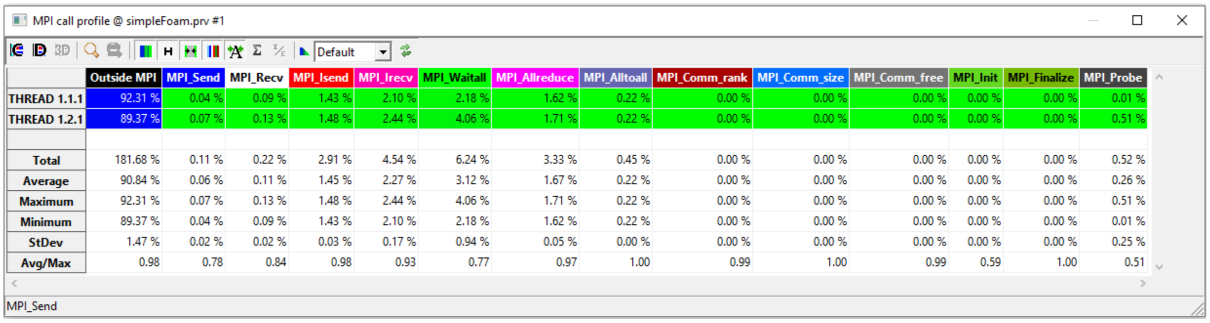


Figure 5. MPI calls view for scenario 1

Looking at the MPI calls profile by percentages, it correlates with the previous assumption as the *MPI_Waitall* call only represents a 2-4% of the total time spent. Overall, the MPI communication takes about 10% of the time in total.



MPI call profile @ simpleFoam.prv #1

Default

| | Outside MPI | MPI_Send | MPI_Recv | MPI_Isend | MPI_Irecv | MPI_Waitall | MPI_Allreduce | MPI_Alltoall | MPI_Comm_rank | MPI_Comm_size | MPI_Comm_free | MPI_Init | MPI_Finalize | MPI_Probe |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| THREAD 1.1.1 | 92.31 % | 0.04 % | 0.09 % | 1.43 % | 2.10 % | 2.18 % | 1.62 % | 0.22 % | 0.00 % | 0.00 % | 0.00 % | 0.00 % | 0.00 % | 0.01 % |
| THREAD 1.2.1 | 89.37 % | 0.07 % | 0.13 % | 1.48 % | 2.44 % | 4.06 % | 1.71 % | 0.22 % | 0.00 % | 0.00 % | 0.00 % | 0.00 % | 0.00 % | 0.51 % |
| | | | | | | | | | | | | | | |
| Total | 181.68 % | 0.11 % | 0.22 % | 2.91 % | 4.54 % | 6.24 % | 3.33 % | 0.45 % | 0.00 % | 0.00 % | 0.00 % | 0.00 % | 0.00 % | 0.52 % |
| Average | 90.84 % | 0.06 % | 0.11 % | 1.45 % | 2.27 % | 3.12 % | 1.67 % | 0.22 % | 0.00 % | 0.00 % | 0.00 % | 0.00 % | 0.00 % | 0.26 % |
| Maximum | 92.31 % | 0.07 % | 0.13 % | 1.48 % | 2.44 % | 4.06 % | 1.71 % | 0.22 % | 0.00 % | 0.00 % | 0.00 % | 0.00 % | 0.00 % | 0.51 % |
| Minimum | 89.37 % | 0.04 % | 0.09 % | 1.43 % | 2.10 % | 2.18 % | 1.62 % | 0.22 % | 0.00 % | 0.00 % | 0.00 % | 0.00 % | 0.00 % | 0.01 % |
| StDev | 1.47 % | 0.02 % | 0.02 % | 0.03 % | 0.17 % | 0.94 % | 0.05 % | 0.00 % | 0.00 % | 0.00 % | 0.00 % | 0.00 % | 0.00 % | 0.25 % |
| Avg/Max | 0.98 | 0.78 | 0.84 | 0.98 | 0.93 | 0.77 | 0.97 | 1.00 | 0.99 | 1.00 | 0.99 | 0.59 | 1.00 | 0.51 |

MPI_Send

Figure 6. MPI calls view by percentage for scenario 1

Obviously, where the execution spent most of the time is outside MPI, that is the OpenFOAM code, representing an 89-92% of the total time spent.

Besides that, Extrae was able to translate the files for each process in approximately 1 minute and not even 2 minutes to complete the Extrae merge process. The impact of Extrae in this scenario was negligible.

Overall, the execution in this scenario was very smooth.

## Scenario 2

Using four processes means that all the computational resources of the front-end host must intervene in the execution, as the Raspberries only have one CPU with four cores. Bearing this in mind, it seems difficult that this combination can provide any better results than the previous ones.

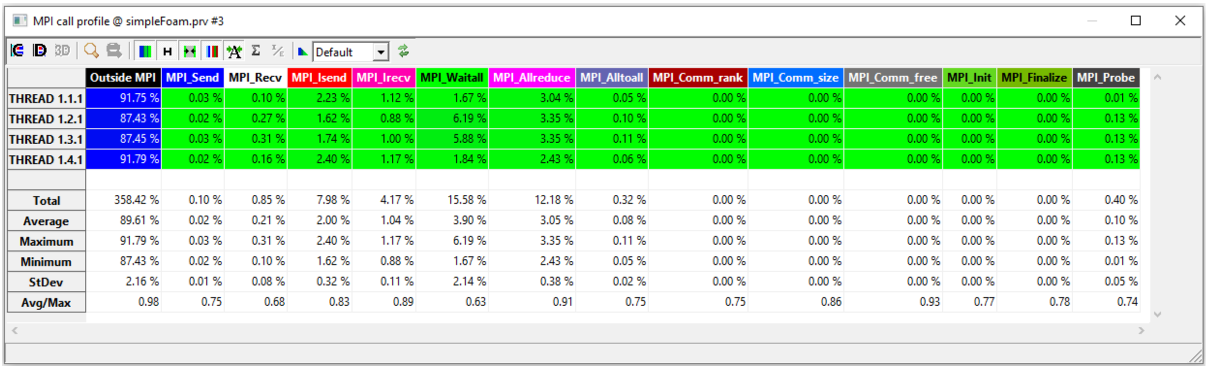
Indeed, this is the scenario that had the worst performance of them all. Not only because the execution of the application took way longer than the previous cases, it took 86 seconds to complete, but also because the process was killed during the Extrae merge process due to resources starvation after 10 minutes.

Subsequent calls of the scenario while having a look at the system resources, revealed that during the Extrae merge process the Random Access Memory (RAM) and the Swap memory were brought to the limits. This is because how Extrae works, that the more processes or threads needs to instrument, the more traces it generates; and obviously, the more traces, the larger the size of the files.

## <u>Scenario 3</u>

This scenario is a remarkably interesting one, because if up to this point the scenario that has given the best results is the scenario one, two processes in a single host, it seems reasonable to think that running four processes in two hosts, two per host, could provide better numbers.

Nevertheless, it took 13 seconds more to complete compared to scenario one, 54 seconds in total.



MPI call profile @ simpleFoam.prv #3

Default

| | Outside MPI | MPI_Send | MPI_Recv | MPI_Isend | MPI_Irecv | MPI_Waitall | MPI_Allreduce | MPI_Alltoall | MPI_Comm_rank | MPI_Comm_size | MPI_Comm_free | MPI_Init | MPI_Finalize | MPI_Probe |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| THREAD 1.1.1 | 91.75 % | 0.03 % | 0.10 % | 2.23 % | 1.12 % | 1.67 % | 3.04 % | 0.05 % | 0.00 % | 0.00 % | 0.00 % | 0.00 % | 0.00 % | 0.01 % |
| THREAD 1.2.1 | 87.43 % | 0.02 % | 0.27 % | 1.62 % | 0.88 % | 6.19 % | 3.35 % | 0.10 % | 0.00 % | 0.00 % | 0.00 % | 0.00 % | 0.00 % | 0.13 % |
| THREAD 1.3.1 | 87.45 % | 0.03 % | 0.31 % | 1.74 % | 1.00 % | 5.88 % | 3.35 % | 0.11 % | 0.00 % | 0.00 % | 0.00 % | 0.00 % | 0.00 % | 0.13 % |
| THREAD 1.4.1 | 91.79 % | 0.02 % | 0.16 % | 2.40 % | 1.17 % | 1.84 % | 2.43 % | 0.06 % | 0.00 % | 0.00 % | 0.00 % | 0.00 % | 0.00 % | 0.13 % |
| | | | | | | | | | | | | | | |
| Total | 358.42 % | 0.10 % | 0.85 % | 7.98 % | 4.17 % | 15.58 % | 12.18 % | 0.32 % | 0.00 % | 0.00 % | 0.00 % | 0.00 % | 0.00 % | 0.40 % |
| Average | 89.61 % | 0.02 % | 0.21 % | 2.00 % | 1.04 % | 3.90 % | 3.05 % | 0.08 % | 0.00 % | 0.00 % | 0.00 % | 0.00 % | 0.00 % | 0.10 % |
| Maximum | 91.79 % | 0.03 % | 0.31 % | 2.40 % | 1.17 % | 6.19 % | 3.35 % | 0.11 % | 0.00 % | 0.00 % | 0.00 % | 0.00 % | 0.00 % | 0.13 % |
| Minimum | 87.43 % | 0.02 % | 0.10 % | 1.62 % | 0.88 % | 1.67 % | 2.43 % | 0.05 % | 0.00 % | 0.00 % | 0.00 % | 0.00 % | 0.00 % | 0.01 % |
| StDev | 2.16 % | 0.01 % | 0.08 % | 0.32 % | 0.11 % | 2.14 % | 0.38 % | 0.02 % | 0.00 % | 0.00 % | 0.00 % | 0.00 % | 0.00 % | 0.05 % |
| Avg/Max | 0.98 | 0.75 | 0.68 | 0.83 | 0.89 | 0.63 | 0.91 | 0.75 | 0.75 | 0.86 | 0.93 | 0.77 | 0.78 | 0.74 |

Figure 8. MPI calls view by percentage for scenario 4

Looking at the results, there are two things that can be highlighted. The first one is that the time spent outside of MPI is very similar, if not better, than in the first scenario (that represented an 89-92% of the total time spent). The second one is that the amount of time that the processes have spent waiting (*MPI_Waitall* call) has slightly degraded for two of the processes. One of the reasons could be network latency on the communication between the hosts, although this reason alone probably is not enough to explain the increase in time.

However, it is important to highlight that the Extrae merge process completed successfully. Since the two hosts participated in the process, the necessary amount of RAM memory available for each host was reduced considerably.

In this case, Extrae was able to translate the files for each process in approximately 6 minutes and the Extrae merge process took about 8 minutes, which are considerably higher times than in the other scenarios.

# 9. CONCLUSIONS

In the beginning of this document, when talking about scientific applications, it was mentioned that reproducibility is one of the core principles in any formal research and it is often expected that the findings of any study can be replicated with a high degree of reliability. This assumption remains true, and it has been successfully fulfilled in the analysis conducted in this study, as in order to compare the scenarios fairly and to be able to provide objective conclusions of the results, the baseline for all of them must be invariant.

It has been demonstrated how distributed scientific applications can be parallelized to solve complex problems. Additionally, it has been observed how the domain of the scientific application can be broken down into smaller tasks to run it concurrently leveraging the MPI programming paradigm.

In this sense, one of interesting fact observed is that not always more computational resources provide better results, and that it is important to find the right balance between resources and performance or even costs. Due to this, instrumentation tools play a key role in the process of understanding the usage that applications do of the resources of a cluster.

Besides that, there is other notorious takeaways. For instance, how containers can help us to simplify processes. Looking back to understand what are the pieces of work that consumed most of the time, it clearly comes to the mind the compilation process of each of the tools and the investigation to figure out their configuration settings on runtime; as all of them have different compile instructions, settings, environment variables, dependencies, pitfalls, etc.

Instead, now this has been a one-off thing necessary to create the image, that gives us the ability to run it as many times as necessary with the flexibility of using any number of MPI processes we are interested in. Not only that, but also it would be possible to run the application in a completely different cluster with little effort.

Additionally, on a production environment, furthermore if we talk about supercomputers, it is safe to assume that resources are allocated randomly by a workload manager, for example Slurm, and you don’t have access to the underlying system at all. In the event of complex applications that have many dependencies between them and that require some pre-setup, containers can be an excellent solution.

Other advantages of using containers are that you cannot mess it up. Containers generally are immutable, and when they are not, the changes only are temporary and are discarded at the end of the lifecycle.

Aside from the containers and their advantages, it is worth taking a moment to think about what could have been the next steps or improvements to the solution that was outlined in this document. Particularly, regarding the Paraver traces generated and the analysis of the results.

Some of the enhancements that could be made are compiling OpenFOAM in Debug mode, to better understand what the application is doing when it is outside of MPI, which in the scenarios described here was about 90% of the time and instrument the cluster at a lower level. For instance, use a Performance Application Programming Interface[24] (PAPI), to see the microprocessor events, or enable the traces on Extrae to measure the performance and usage of disk input/output, memory usage, network latency, context switching, etc.

Doing a deep and thorough analysis of the behaviour of a scientific application is a task that can take long time, the order of weeks or months, and that requires a lot of expertise to identify patterns or trends.

[24] Link to https://icl.utk.edu/papi/

## 10. BIBLIOGRAPHY

Iserte Agut, Sergio; Catalán Pallarés, Sandra; Carratalá Saez, Rocío; López Huguet, Sergio (2021). *Construya su propio supercomputador con Raspberry Pi.*

Supriya Saxena (2021). *ARM processor and its features.* Read in https://www.geeksforgeeks.org/arm-processor-and-its-features in April 2022.

Barcelona Super Computing Center tools website (2022). Read in https://tools.bsc.es in May 2022.

University of Edinburgh, UK (2021). *Running MPI parallel jobs using Singularity containers*. Read in https://epcced.github.io/2021-07-29_Singularity_Online/08-singularity-mpi/index.html in May 2022.

VMware (2022). *What is a virtual machine?* Read in https://www.vmware.com/topics/glossary/content/virtual-machine.html in June 2022.

Ian Buchanan, Atlassian (2022). *Containers vs virtual machines.* Read in https://www.atlassian.com/microservices/cloud-computing/containers-vs-vms in June 2022.