Haute école d'ingénierie et d'architecture Fribourg
Hochschule für Technik und Architektur Freiburg

Bachelor of Science HES-SO in Engineering
Bd de Pérolles 80
CH-1700 Fribourg

Bachelor of Science HES-SO in Engineering

Department: Computer Science and Communication Systems

Evaluation et test des capacités de calcul des différents clusters de l'école avec Terraform et Kubeflow

Author:
Martin Roch-Neirey

Under the supervision of:
Sébastien Rumley
Beat Wolf

Written as part of Semester 5 project

Fribourg, HES-SO//Bachelor, February 1, 2024

History

Version	Date	Modification
0.0.0	2023.10.19	Document creation
0.0.1	2023.10.29	Analysis - Terraform and Ansible
0.0.2	2023.11.01	Analysis - HPL and MPI
0.0.3	2023.11.03	Analysis - OpenStack
0.0.4	2023.11.07	Analysis - Kubeflow and Kubernetes
0.1	2023.11.15	End of analysis
0.1.1	2023.11.25	Design - infrastructures
0.1.2	2023.11.29	Implementation - benchmark
0.2.0	2023.12.07	End of design
0.2.1	2023.12.12	Implementation - automation processes
0.2.2	2024.01.09	End of implementation
0.2.3	2024.01.15	Results - Raw results
0.3.0	2024.01.24	Rework of design and implementation. More on results
0.3.1	2024.01.28	Results - other experiments
0.4.0	2024.01.31	End of results
1.0.0	2024.02.01	Writing of conclusion and final review

Table 1: Version history

Contents

List of Figures

List of Tables

1 Introduction

This document is the final report of the semester project entitled "Evaluation et test des capacités de calcul des différents clusters de l'école avec Terraform et Kubeflow", carried out as part of a semester 5 project at the Haute Ecole d'Ingénierie et d'Architecture of Fribourg. This work was produced by Martin Roch-Neirey.

1.1 Context

The HEIA-FR has several computer server clusters used by applied research institutes, professors and students. They are used to virtualise computers, deploy workloads as containers and other applications. As part a 16-week semester project, this project aims to measure and compare the computing power of these different clusters. The technologies concerned are VMware, OpenStack and Kubernetes. As the title says, suggested tools are Terraform and Kubeflow.

Many benchmarks exists to calculate theorical computing capacity of one or more computer. One of them is the High-Performance Linpack benchmark, whose objective is to solve systems of linear equations by the form $ax = b$ and other mathematical problems. It is possible to execute this benchmark on multiple cores, and hosts, using Message Passing Interface principle. It then allow computers to communicate and split tasks, so that they can focus on one part of the linear equations system. Theoretically, it is possible to divide execution time by n, in case there is n computers with the same computing capacities working in the problem. But in reality, the practical results may differ from the theoretical ones.

As well as comparing the computing capacities of the HEIA-FR various clusters, this project is also a good way to understand which of the clusters seems to be best suited to high-performance computing. To do this, the benchmark results are processed and analysed so that each cluster obtains a 'scaling efficiency' score, in addition to other analyses. Finally, this project also aims to compare different clusters with a modern laptop: some questions are often asked like "If this problem takes 2 hours to be solved on my laptop, how many time should it take on the VMware cluster ?", and this project should be able to give an answer.

1.2 Objectives

This project aims to estimate computing capacities of different types of clusters inside of the HEIA-FR. These clusters are used by different institutes, professors and students, and should be able to carry large amount of computing tasks.

The main objectives of this project are:

- Measure and compare the computing power of VMware, OpenStack green and Kubernetes clusters.
- Know how clusters and blades behave when they're under maximum stress.

In addition to these key objectives, there is one secondary objective:

- Analyze eletrical comsuption of blades depending on compute tasks running on them.

1.3 Structure and writing conventions of this report

This report is organized in 6 chapters, from the introduction to the conclusion. The reader is invited in the chapter 2 to discover how the HPL benchmark works, and what technologies, tools and clusters are involved in this project. Next, the chapter 3 shows the design of the experiments, with a focus on each cluster. The chapter 4 explains how the benchmark is implemented technically, for a native, virtualized and containerized environment. If applicable, this chapter also explains how the benchmark has been automated. The penultimate chapter shows the results collected and the conclusions drawn, but also lists some other experiments that have not been finished yet. Finally, chapter 6 presents the overall conclusion of the project.

At the beginning of each chapter, its structure is explained. Some of the technical vocabulary is explained in the glossary at the end of this document. Each word inserted in the glossary is underlined the first time it appears in the document.

2 Analysis

This section reports to the analysis phase made in this project. This phase is significant as it helps to choose the tools that are used in the design and implementation phase.

Many tools exists to automate deployment and configuration of virtual machines and containers. Despite this, it is fundamental to select the right tools wisely, depending on project specifications, technical knowledge, deadlines, and technical stack used.

This chapter is organized in many parts, depending on the section of the project (virtualization or containerization) and the tools involved in it. An analysis of each tool is provided to check if it fits to the project.

An introduction to the benchmark stack is also presented at the start of this chapter.

Contents

2.1 Benchmark stack

The High Performance Linpack benchmark uses different tools to estimate computing capabilities of computers. Each tool can be used in many versions, and compiled using different parameters that may impact the final results.

2.1.1 HPL and LINPACK

High Performance Linpack, often abbreviated as HPL is a widely used benchmark for measuring the floating-point performance of computer systems. It focuses on the performance of solving a dense system of linear equations. The benchmark measures the rate at which a computer can perform a set of linear algebra operations, particularly the solution of a system of linear equations using LU decomposition. The results are expressed in floating-point operation per second (FLOPS) and are typically measured in teraflops (TFLOPS).

HPL is based on the LINPACK software library, which stands for *Linear Algebra Package*. The library is a collection of Fortran subroutines that analyze and solve linear equations problems. Developed in the late 1970s, it was one of the earliest libraries for high-performance computing. LINPACK focuses mainly on solving systems of linear equations and linear least-squares problems using a variety of direct methods, primarily through LU and other factorization methods.

The LU decomposition (lower-upper, also called LU factorization) factors a matrix as the product of two triangular matrixes : one lower, and one upper.[1] LU decomposition uses the Gaussian elimination (also known as row elimination). This method is appreciated by IT people because it is possible to determine the number of operations needed for a given matrix. The formula is shown below :

$$N_{ops} = \frac{2}{3}n^3 + 2n^2$$

Where N_{ops} is the number of operations performed. Therefor, the algorithmic complexity of this implementation of the HPL benchmark is $O(n^3)$. For example, with a square matrix of size 10'000, N_{ops} is equal to:

$$N_{ops} = \frac{2}{3} * 10'000^3 + 2 * 10'000^2$$
$$N_{ops} = 6.67 * 10^{11}$$

$6.67 * 10^{11}$ operations are needed to solve a problem of size 10'000. Next step is to measure the time that a computer takes to solve the problem, and to calculate the number of operations per second. With previous N_{ops} and a computer that solved the problem in 10 minutes, the result is:

$$R_{max} = \frac{6.67*10^{11}}{10*60}$$
$$R_{max} = 1111666667$$
$$R_{max} = 1111666667$$
$$R_{max} = 1.111666667 * 10^9$$

Where R_{max} defines the performance in FLOPS of a machine for the given problem. In this case, the computer can execute 1.111666667 GFLOPS.

2.1.2 OpenBLAS

To execute the benchmark and use the LINPACK library, HPL uses an implementation of a BLAS (Basic Linear Algebra Subprograms) library, called OpenBLAS. BLAS is a specification that prescribes a set of low-level routines for performing common linear algebra operations such as vector addition, scalar multiplication, linear combinations, and matrix multiplication.

OpenBLAS acts as a high-performance backbone for the linear algebra operations in HPL, enabling more efficient and faster computations, which is vital for achieving high FLOPS in the HPL benchmark.

OpenBLAS is typically a tool that must be compiled depending on the architecture of the computer, otherwise the benchmark may not use the full computing capacity of it.

2.1.3 OpenMPI

The HPL is widely used to estimate computing capacities of supercomputers and clusters of computers. To do so, the benchmark divides multiple linear algebra problems and forwards them to many computers. Therefor, the communication between hosts is one of the most important point of the benchmark. Here is why OpenMPI is used in this project:

> *"Today's largest high-performance computers, a.k.a. supercomputers, are all organized around several thousands of compute nodes, which are collectively leveraged to tackle heavy computational problems. This orchestrated operation is only possible if compute nodes are able to communicate among themselves with low latency and high bandwidth."*[2]

To communicate between hosts, HPL uses MPI (Message Passing Interface), which is a standardized and portable message-passing system that allows processes or tasks running on different nodes or processors of a parallel or distributed computing system to communicate with each other. One famous implementation of MPI is OpenMPI, which can be used by HPL.

MPI is based on the fundamental concept of passing messages between processes. These messages can contain data or synchronization information and allow processes to communicate. It is designed for parallel computing, where multiple tasks or processes run concurrently. These processes can be executed on the same computer (communication between cores), or distributed across a cluster of machines (communication between nodes, and cores).

Basically, OpenMPI uses SSH protocol to communicate between nodes. It is then possible to transmit commands, problems specifications and results of the benchmark across multiple computers.

2.1.4 Full stack

To summarize all the tools used by HPL, the complete stack used by the benchmark is shown below :

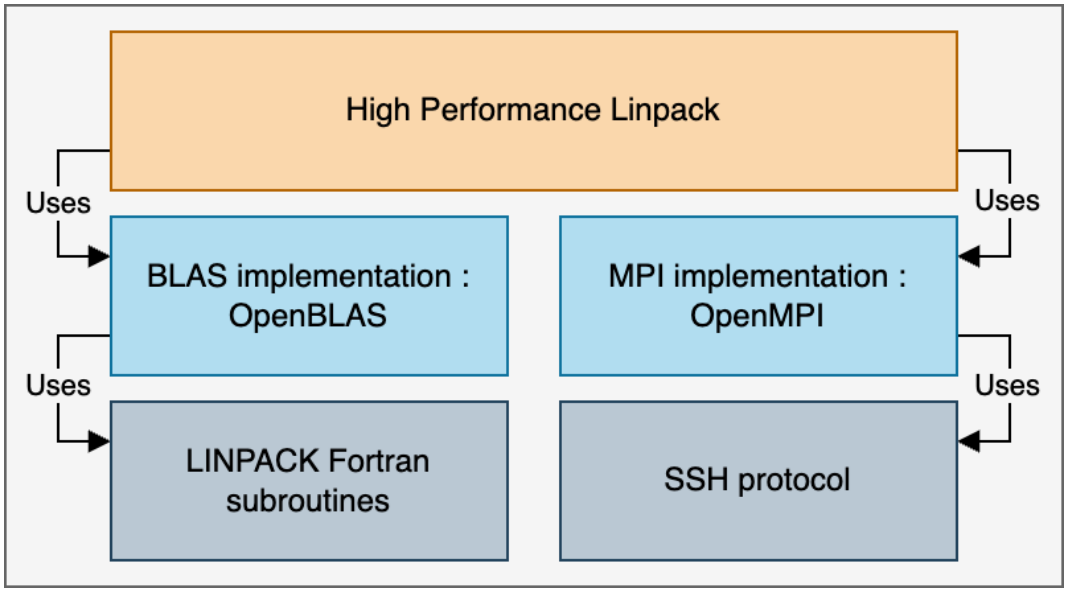


Figure 2.1: Full HPL stack

As shown, the HPL benchmark uses 2 main tools:

1. **One BLAS implementation:** OpenBLAS, to use LINPACK Fortran subroutines efficiently and get higher FLOPS than without the tool.

2. **One MPI implementation:** OpenMPI, to allow communication between all nodes and cores.

2.2 Virtualization

This section aims to describe the virtualization technologies that are involved in this project. The first part presents 2 types of clusters, and then 2 automation tools are presented.

2.2.1 VMware

VMware is a global leader in cloud infrastructure and digital workspace technology. At the heart of VMware's offerings is the vSphere suite, which includes ESXi, a key component that facilitates the virtualization of physical servers.

Figure 2.2: VMware ESXi Logo

People often talk about VMware, vSphere and ESXi without necessarily knowing exactly what the difference is between them. vSphere is VMware's cloud computing virtualization platform. It serves as a complete infrastructure for virtualized data centers, allowing businesses to run, manage, connect, and secure applications in a common operating environment across clouds and devices. vSphere essentially transforms hardware into a shared, aggregated resource, which results in improved efficiency, agility, and flexibility in the deployment of IT services. The HEIA-FR has one vSphere instance, representing the ISC VMware cluster.

ESXi, on the other hand, is a Type 1 Hypervisor that is a part of the vSphere suite. This hypervisor is crucial as it directly runs on the physical server and partitions it into multiple virtual machines (VMs). Each VM can run its own operating systems and applications as if they were running on a distinct physical machine. ESXi's lightweight architecture, which lacks a full operating system, minimizes the resources required for the hypervisor, making it more efficient and secure compared to older, full OS-based solutions.

The characteristics of the ISC VMware cluster are shown below:

- Name: VMware ISC
- Number of blades: 4
- 2 VMware ESXi 7.0.3 (Cisco UCSX-210C-M6)
- 2 VMware ESXi 7.0.3 (Cisco UCSB-B200-M5)
- Total RAM: 2.5TB
- Total CPU: 72
- Total storage: 50TB
- Number of virtual machines running on it before the project: 569

2.2.2 OpenStack

OpenStack is an open-source cloud-computing platform that manages and orchestrates a pool of computing, storage, and networking resources. It enables organizations (like HEIA-FR) to create and manage private or public clouds, offering infrastructure-as-a-service (IaaS) capabilities for deploying and scaling virtual machines, storage, and network resources. It can be considered as a cloud-oriented hypervisor.

Figure 2.3: OpenStack Logo

In fact, OpenStack is essentially a collection of tools that help manage the whole resources (computing, storage, networking...). OpenStack has several key modules, each serving a distinct function. For example, Nova is crucial for managing virtual machines and computing resources, making it the backbone for processing tasks in the cloud. Swift provides object storage, handling data storage in a non-hierarchical format, which is essential for scalability and redundancy. Glance, responsible for managing disk and server images, is important for the rapid deployment of new instances. Neutron offers networking capabilities, ensuring communication between different components in the cloud infrastructure. Keystone plays an important role in security, handling authentication and authorization. Lastly, Horizon offers the web-based interface for managing and interacting with the various OpenStack services. Here is an overview of all the OpenStack modules:

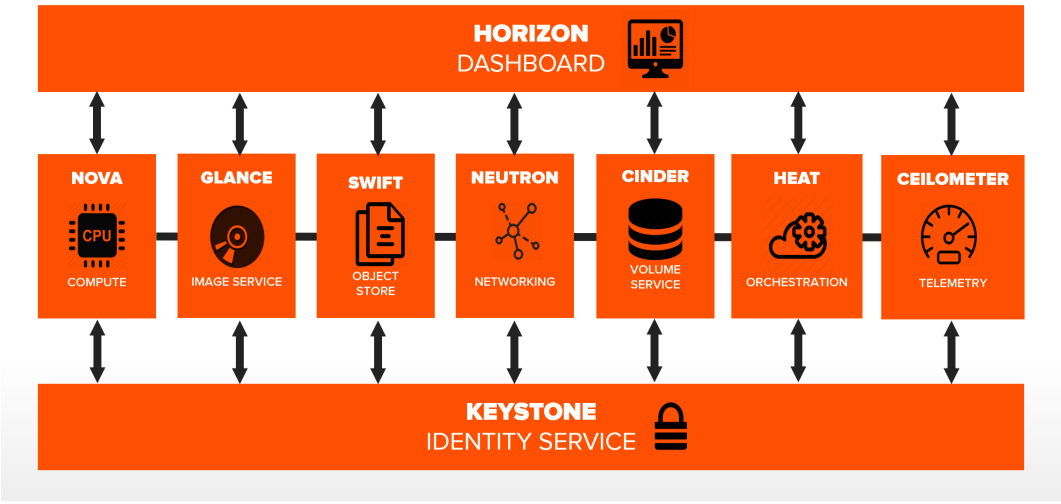


Figure 2.4: Simplified OpenStack modules view (ReadTheDocs)

The HEIA-FR has 2 OpenStack clusters, and the following one is used in this project:

- Name: OpenStack Green
- Number of blades: 6 (ProLiant DL360 Gen9 / Gen10)
- Total RAM: 754GB
- Total vCPU: 192
- Total storage: 70.7TB
- Number of virtual machines running on it before the project: 0

During the project, the green cluster was in development mode, and the only virutal machines emulated on it were for this project.

2.2.3 Terraform

Terraform is an Infrastructure as Code tool developed by Hashicorp, and published in it's first version in 2014. It allows it's users to define an infrastructure in documents, and to apply the configuration to an hypervisor to build virtual machines, virtual networks... Terraform is well suited for cloud computing projects as it is possible to choose flavor of virtual machines, create virtual networks and routers, add an SSH key to enable secure distant connection and apply security rules to machines (e.g. to only allow SSH port).

Terraform is able to connect to many cloud providers API, including VMware and OpenStack. These technologies and projects are using different APIs, with different versions and capabilities. Each cloud provider has a module on Terraform called *provider*. In this project, the OpenStack provider is the only used as the virtual machines on the VMware cluster are already up and running.

2.2.4 Ansible

Ansible is a configuration automation tool created by Michael DeHaan, and now maintained by Red Hat.

Ansible uses SSH protocol to connect to distant machines to configure them. The tool needs an access with an SSH key configured on machines to be allowed to connect. As written in the Terraform analysis, virtual machines can be provisioned with a pre-registered SSH key. This SSH key allows Ansible to configure these machines, and possibly add an other SSH key to authorize an administrator distant connection. Ansible control node can be executed on any machine running Python. Unlike tools like Cheff or Puppet, Ansible doesn't need to have an agent installed on every machine. The SSH protocol is all it needs to configure machines.

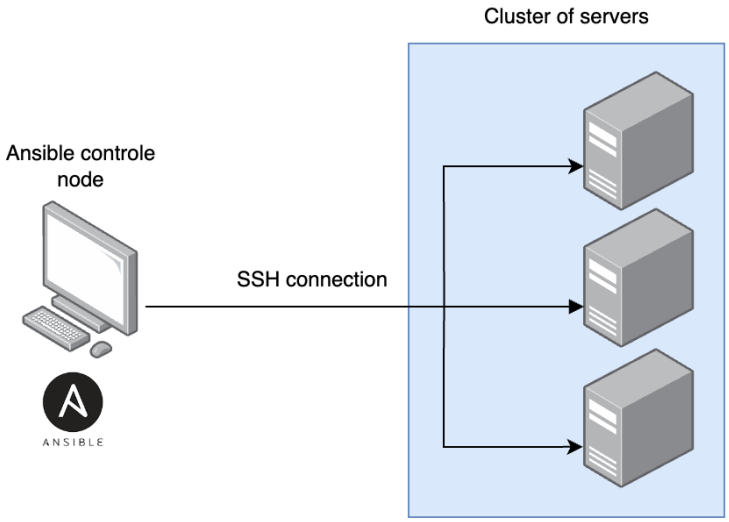


Figure 2.5: Ansible controls nodes only with SSH protocol

Ansible has different types of files, that are used to automate configuration, here are the most important:

- Inventories: lists the hosts (target machines) on which Ansible is going to perform operations. Hosts can be stores as groups, subgroups... It is also possible to define variables for a specific group.
- Playbooks: YAML files that contain a series of tasks to be executed on the hosts. They describe the desired state of the system and are used to automate configuration of machines.
- Modules: programs that run on target hosts to perform specific tasks. Ansible includes many built-in modules for managing various aspects of the system, such as file management, software configuration, user management, and more.
- Roles: collections of playbooks, variables, and files that help organize and reuse Ansible code. Roles simplify configuration management by structuring the code in a modular way.

In this project, Ansible is used to configure virtual machines, deploy and configure the HPL stack on them, but also to execute and retrieve benchmark results automatically.

2.3 Containerization

This section reports on the various tools and platforms used for containerization. As the iCoSys institute has a Kubernetes cluster, it is interesting to estimate its computing power and compare it with others.

2.3.1 Kubernetes

Kubernetes is a well-known and powerful workloads orchestrator, most of the time using the Docker engine (but not exclusively) to execute containers. The tool is able to automate deploying, scaling and operating multiple containerized applications.

Figure 2.6: Kubernetes logo

By abstracting the hardware infrastructure layer, Kubernetes allows the deployment of various workloads, from stateless to stateful applications, including databases. This versatility, combined with a robust ecosystem, has led to widespread adoption in the IT industry. A significant feature of Kubernetes is its ability to maintain the desired state of applications. For example, if a container fails, Kubernetes can automatically replace it, providing a form of self-healing crucial in production environments.

The iCoSys applied research institute, which is very active in the field of artificial intelligence and complex systems, has several computing clusters, including a Kubernetes cluster.[3] This cluster is used in this project.

2.3.2 Kubeflow

Kubeflow is an open-source platform designed to facilitate the development, orchestration, deployment, and running of machine learning workflows on Kubernetes. It provides a straightforward way to deploy open-source systems for ML to diverse infrastructures. Essentially, Kubeflow makes deployments of machine learning workflows on Kubernetes simple, portable, and scalable. In fact, this is one of their main arguments for using their tool: "Anywhere you are running Kubernetes, you should be able to run Kubeflow"[4].

Figure 2.7: Kubeflow logo

The Kubeflow architecture has many components, including "operators". These operators can train different ML models on the Kubernetes cluster. Kubeflow has different operators such as TensorFlow, PyTorch, MPI Training and others. The idea is to use the MPI Training operator to perform the benchmark on the Kubernetes cluster, without having to deploy an implementation of OpenMPI. Actually, it has been discovered that this is not possible. Kubeflow's MPI Training operator only purpose is to train ML models. The project must then implement it's own MPI library, using OpenMPI. The entire benchmark stack must then be containerized to be deployed on the iCoSys Kubernetes cluster, and it is necessary to ensure that each pod can access others over SSH.

3 Design

This chapter takes the reader through the technical design of the project, including the infrastructure set up in each cluster and the benchmark configuration. The HPL benchmark has a file called *HPL.dat* which is used to configure the problem size, and how it will be splitted among all available cores. The results of the benchmark partially depends on the benchmark configuration and the BLAS library used. It is a long process to find the good configuration, depending on the architecture of the super-computer, operating system... In this project, the same configuration has been used on each cluster (as on the modern laptop).

A website[5] allows benchmark administrators to enter their system specifications, and the output is an *HPL.dat* file which is supposed to be well-configured for the given configuration. As the main objective of this project is to compare different clusters, one configuration has been chosen and applied to all clusters and modern laptop.

Contents

3.1 Benchmark configuration

As the *HPL.dat* file is pushed on the benchmark node using an Ansible playbook, it allows the benchmark administrator to choose the value for each parameter locally, and then apply the modifications with a single command. This part is described in the implementation chapter. The following configuration has been chosen for the whole project:

- Problem size (N): 28800 (limited by RAM size of each virtual machine)
- Process raws (P): 1 (voluntarly set to 1 so only Q will change)
- Process columns (Q): *NP* (with *NP* the number of cores allocated to compute the problem)
- Block sizes (NBs): 200 (chosen arbitrarily)
- All others parameters are the default ones provided in the HPL portable implementation offered on the netlib website[6].

It is also important to note that the OpenBLAS library has been compiled in a "non-threaded" version, so that each xhpl process launched cannot be executed by multiple threads, but only by the main one. Some experiments has been done with the threaded version, and the results were skewed and did not reflect the real computing capacity of the clusters. More information are given on this point in the implementation chapter.

3.2 Benchmark execution procedures

To compare the clusters and their respective computing power, it is interesting to compare them in different situations. For this reason, a precise test procedure has been defined. The output is a graph showing the benchmark's result in GFLOPS depending on the cores allocated to the problem. By sticking one HPL process (also called *xhpl process*) to one core, the graph can then show the GFLOPS depending on the number of xhpl processes executed. These graphs are shown and explained in the chapter 5.

When having multiple nodes executing the benchmark, it is possible to distribute xhpl processes on all of them, even if only one node could handle the full problem. For example, if there are 4 nodes, each with 8 cores and the number of xhpl processes is 8:

- It it possible to execute all xhpl processes on one node.
- It is also possible to distribute evenly the processes, by assigning 2 xhpl processes on each node.
- But it is also possible to choose any other combination where each xhpl process is assigned to a core, no matter the node executing it.

To ensure the experiments case are the same for each cluster, and for the sake of simplicity, it was decided to use all the cores of a node before moving on to the next one. This also ensure that each cluster is tested gradually and this is a good way to highlight losses due to the network usage. Due to quotas availability on the clusters, the following nodes were emulated for the benchmark:

- 4 virtual machines with 8vCPU and 8GB RAM, for the VMware and OpenStack clusters.
- 4 pods running on different workers for the Kubernetes cluster.

The distribution of processes on the nodes in accordance with the test procedure is defined in the appendix A. This way of running the benchmark fits perfectly into the strong scaling box, which assesses how efficiently a system can decrease the processing time for a fixed-size problem by proportionally increasing the number of processing elements.

3.3 Infrastructure design

This section presents the different infrastructures implemented on the clusters, with the specific features of each. It is important to note that the benchmark results may depend on the blades on which the virtual machines are located. If all the virtual machines are located on a single blade and that blade does not have enough physical cores, it will be in an overprovisioning situation and will have to juggle between the different emulated machines, impacting overall performance. This notion was not evaluated at the beginning of the project, and it is typically not possible to guarantee on each cluster that each virtual machine is on a different blade.

3.3.1 Specific VMware design

There are 4 virtual machines available for the benchmark on the ISC VMware cluster. They all have the same specifications and are running an Ubuntu 22.04 LTS operating system.

The virtual machines were emulated on 2 different environments, during 2 phases of the project:

- On an old blade (or multiple blades) of the cluster, before the mi-semester. It has not been thinked to retrieve the location of the virtual machines and the blade model concerned at this time.
- After the mi-semester, the VMware cluster got 2 old blades switched with new ones. From this time, the 4 virtual machines were emulated on one of the new ones (Cisco UCSX-210C-M6 model).

The VMware benchmark infrastructure is shown below:

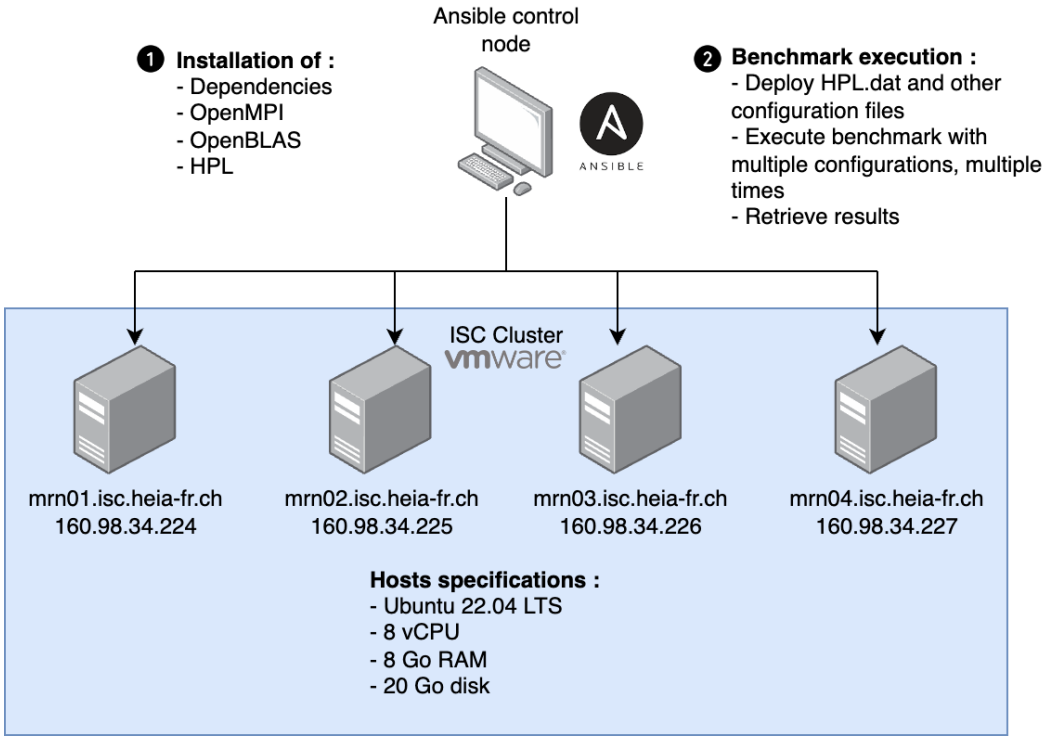


Figure 3.1: Infrastructure deployed on the ISC VMware cluster for the benchmark

Note that these 4 virtual machines were deployed by one of the administrators of the cluster, the Terraform API access has not been granted for this project.

3.3.2 Specific OpenStack design

The entire OpenStack green cluster of the HEIA-FR was made available for this project, allowing the instanciation of multiple virtual machines with a lot of configurations possibilities. However, as the main objective of this project is to compare different clusters, it has been decided to create 4 virtual machines with the same specifications as those on the VMware cluster. This time, the Terraform API has been used to provision the infrastructure. The benchmark can then be fully automated from an empty OpenStack project to the final results (the next chapter explains how the automated process works). The following image shows the infrastructure deployed on the OpenStack cluster:

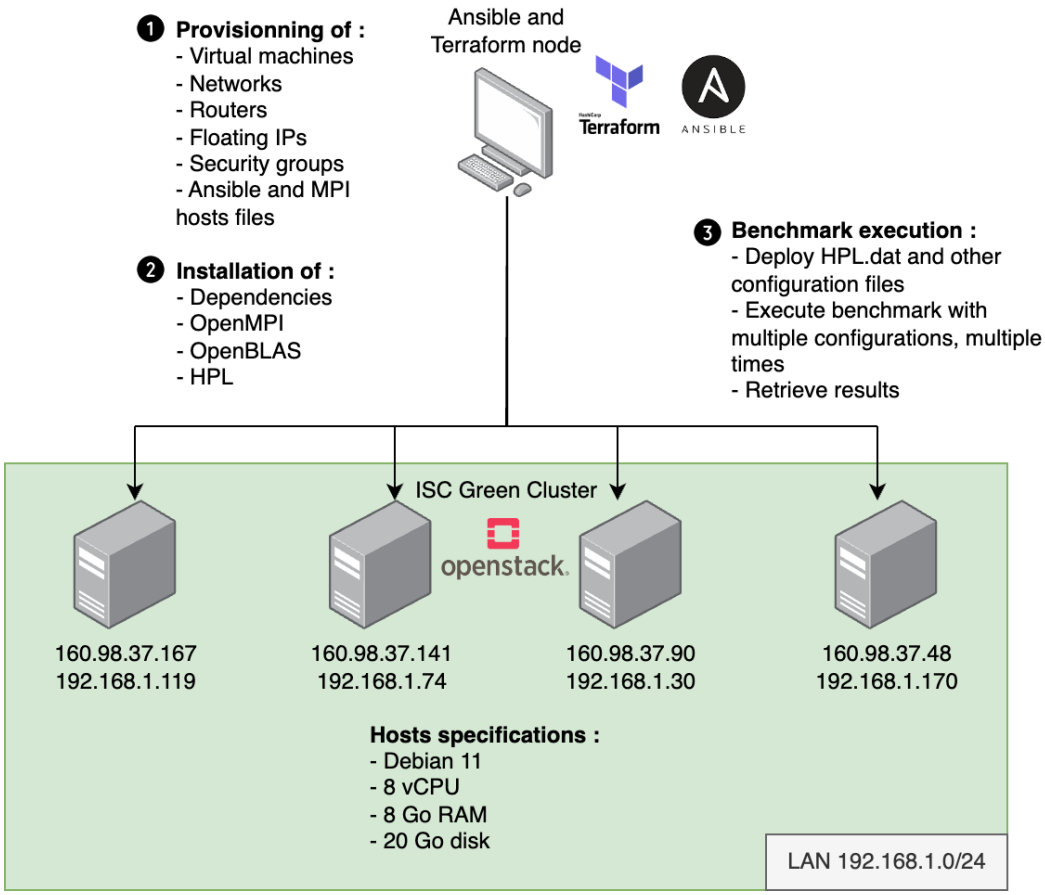


Figure 3.2: Infrastructure deployed on the ISC green OpenStack cluster for the benchmark

This schema shows the entire automated process, from an empty OpenStack project to the results of the benchmark. If the administrator wants to perform the benchmark on a different infrastructure, it is just necessary to change the number of virtual machines in a Terraform file. The Ansible and MPI hosts files will automatically be updated when provisionning the new version of the infrastructure. Each virtual machine is accessible via a floating IP, which is not mandatory but better to perform some monitoring tasks. The Terraform pipeline populates the Ansible and MPI hosts with the floating IPs, but this can be changed easily.

3.3.3 Specific Kubernetes design

The iCoSys Kubernetes cluster has 8 worker nodes and 3 master nodes[3], it has been decided to use PodAffinity rules to deploy the 4 pods on different workers. No Terraform is needed here as the Kubernetes platform is considered as a production environment, up and running since many years. The Kubernetes part wasn't completely automated because each pod is privately addressed, and a bit more configuration would have been needed to expose a pod (with more time, the first solution implemented would have been to use a NodePort configuration). The benchmark stack is implemented in the Docker image so that they are directly usable. The Kubernetes schema is shown below:

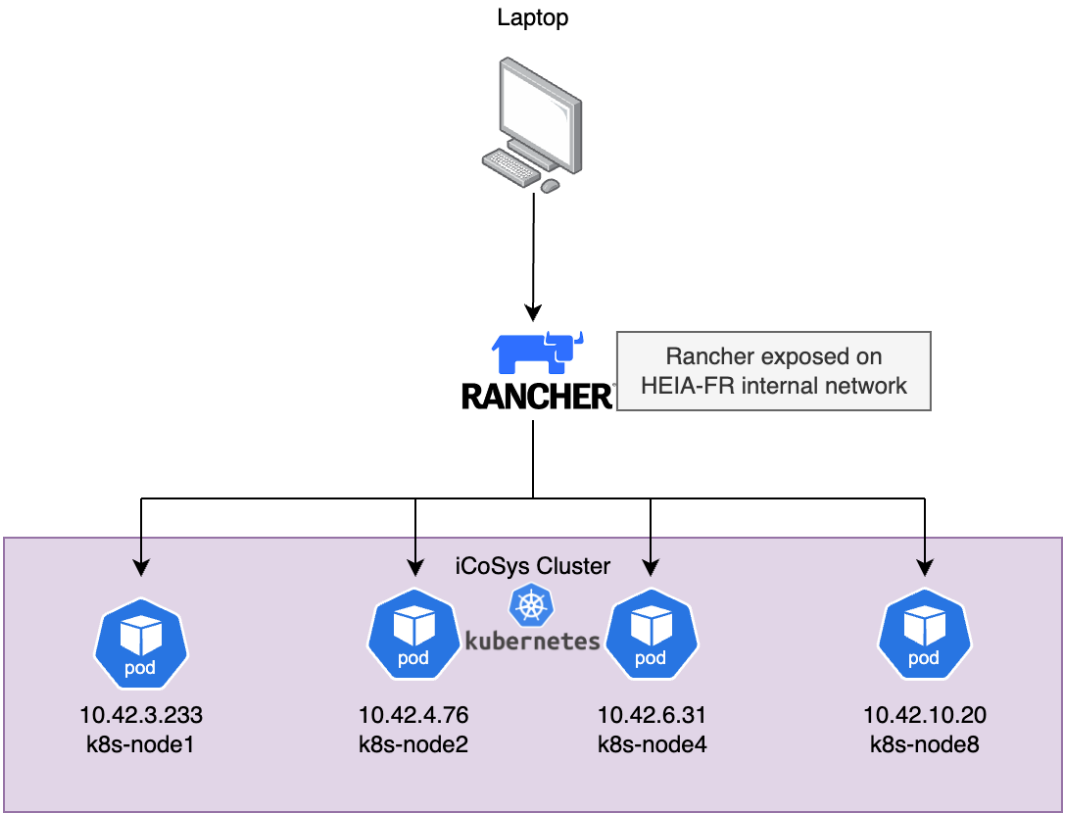


Figure 3.3: Infrastructure deployed on the iCoSys Kubernetes cluster for the benchmark

The Docker image build process is explained in the implementation chapter, as well as the configuration files used on the Kubernetes cluster.

4 Implementation

This chapter explains how the project was technically implemented. The first part is about the compilation of the benchmark on a modern laptop, and then the focus is made on the virtualization and containerization parts. As explained in the design chapter, the BLAS library used and the benchmark configuration are the same for all clusters and the modern laptop. The first part really focuses on how the benchmark is compiled, and the second and third parts explains more how the benchmark has been automated, and deployed on multiple environments.

This chapter does not aim to provide a complete tutorial on how to download, compile and use the benchmark. It outlines the major parts of the implementation, but it is suggested to visit the GitLab technical repository of the project to see the full code, playbooks and configuration files[7].

Contents

4.1 Native benchmark compilation

The benchmark has been compiled on an ARM architecture first, using a MacBook Pro M2. The way the compilation has been done on this laptop is the same for the clusters, as compilators adapt their instructions depending on the architecture.

4.1.1 OpenMPI

The first thing to have is a Java installation. A Java Development Kit may be mandatory as the OpenMPI library is going to be configured and compiled with Java flags, and the Java Runtime Environment does not offer all tools required. Depending on the operating system, this can be done with a package manager (*apt*,*brew*...) or via the official website[8]. It is also necessary to have a Fortran and C compilers (*gfortran* and *g++* are the one used here).

Next, the OpenMPI library can be downloaded on the official website[9]. The version used here is 4.1.6. The library must be configured before compiled, and this is done with the following command[10]:

```
./configure \
--enable-mpi-java \
--with-jdk-bindir=$JAVA_HOME/bin \
--with-jdk-headers=$JAVA_HOME/include/ \
--prefix=/home/pi/builds/openmpi/
```

The *JAVA_HOME* environment variable must point to the home directory of the java installation, most of the time under */usr/lib/jvm/default-java*. Once configured, OpenMPI can be compiled using the following commands:

```
make -j 4
make install
```

The first command is used to compile the library using 4 "jobs", to reduce the compilation time. The second command is used once the library is compiled, to install it. OpenMPI is now ready to use.

4.1.2 OpenBLAS

The first step to install the OpenBLAS library is to download it from the official website[11]. The version used here is 0.3.24. Once downloaded, it can be compiled using specific flags given here:

```
make -j DYNAMIC_ARCH=0 CC=gcc FC=gfortran HOSTCC=gcc \
BINARY=64 INTERFACE=64 NO_AFFINITY=1 NO_WARMUP=1 \
USE_OPENMP=0 USE_THREAD=0 USE_LOCKING=1 LIBNAMESUFFIX=nonthreaded
make LIBNAMESUFFIX=nonthreaded install
```

These flags are mandatory to ensure that OpenBLAS is compiled in a non-threaded version. Otherwise, the benchmark processes will launch threads and they will be distributed among all cores, even if the benchmark must be executed on some specific

cores. Note that the *LIBNAMESUFFIX* flag is not mandatory but is a good way to know that this is the non threaded compiled version. This has been a problem during the project, and the solution has been found on a GitHub repository[12].

4.1.3 HPL

The final step is to download, compile and install the HPL benchmark. The implementation used in this project can be found on the netlib official website[6]. The version used here is 2.3. To compile the benchmark, it is necessary to fill a Makefile with specific information like OpenBLAS and OpenMPI builds locations, the Fortran and C compilers to use, and the flags to pass to these compilers. The Makefile can differ from an operating system to another, as the locations and other configurations may be different. The complete Makefile is available on the technical GitLab project[7].

Once the Makefile is configured, the following command will compile the benchmark.

```
make arch=linux64
```

Note that *linux64* is the extension of the Makefile previously modified. In fact, the full name of this file is *Make.linux64*. It can be whatever else, but it must be specified when compiling the benchmark so that the compilers use the right configuration file. Once the compilation is done, an executable file called *xhpl* and a text file called *HPL.dat* should be available at *[HPL directory]/bin/linux64/* folder.

4.1.4 Execute the benchmark

To execute the benchmark, there are 2 things to do:

1. Configure the benchmark as needed, in the *HPL.dat* file.
2. Start the benchmark with the following command while being in the *[HPL directory]/bin/linux64/* folder:

   ```
   mpirun -np X xhpl
   ```

   With X the number of MPI ranks that are going to be launched. X must also be equal to $P * Q$ defined in *HPL.dat* file.

The given command only starts the benchmark on one node. To perform the benchmark on a cluster of hosts, there are 2 ways:

1. Passing the hosts in CLI, with the command.
2. Using a MPI hosts file.

This project uses an MPI hosts file to list the nodes that are going to participate. Here is an example of this kind of file:

```
160.98.37.167 slots=8
160.98.37.141 slots=8
```

Each line represents a host by its IP address, and the number after the slots variable is equal to the number of cores available on that machine. In the virtualization part of this report, it is explained how this file is generated automatically by Terraform. In the case that this kind of file is used, the following command is used to start the benchmark:

```
mpirun --hostfile [filename] -np X xhpl
```

Once the benchmark has been done, the following result should appear on screen (or in a file, it the command has been piped in an output file):

```
================================================================================
T/V                N    NB     P     Q               Time                 Gflops
--------------------------------------------------------------------------------
WR00L2L2       28800   200     1     1             261.22             6.0969e+01
HPL_pdgesv() start time Tue Jan 23 08:36:16 2024

HPL_pdgesv() end time   Tue Jan 23 08:40:37 2024

--VVV--VVV--VVV--VVV--VVV--VVV--VVV--VVV--VVV--VVV--VVV--VVV--VVV--VVV--VVV-
Max aggregated wall time rfact . . . :               3.10
+ Max aggregated wall time pfact . . :               0.55
+ Max aggregated wall time mxswp . . :               0.15
Max aggregated wall time update  . . :             257.76
+ Max aggregated wall time laswp . . :              11.49
Max aggregated wall time up tr sv  . :               0.35
--------------------------------------------------------------------------------
Ax-b_oo/(eps*(A_oo*x_oo+b_oo)*N)=   2.15657916e-03 ...... PASSED
================================================================================

Finished      1 tests with the following results:
              1 tests completed and passed residual checks,
              0 tests completed and failed residual checks,
              0 tests skipped because of illegal input values.
--------------------------------------------------------------------------------

End of Tests.
================================================================================
```

The effective result of the benchmark is under the "Gflops" word. Here, as an example, it is approximately 61 GFLOPS.

4.2 Virtualization and benchmark automation

This part describes how the benchmark has been automated, to be able to perform a benchmark from an empty OpenStack project with only 3 commands. Note that all the Terraform files created are only working with the OpenStack Terraform provider. It has not been needed to develop a Terraform script for the VMware cluster as the virtual machines were provided by an administrator. The VMware part is still automated,

using Ansible, exactly like the OpenStack. Ansible playbooks are fully compatible with the Kubernetes pods that runs the benchmark, but at least one pod must be publicly accessible, which has not been done in this project.

This part of the report makes a lot of references to code files, which are fully accessible on the technical GitLab repository of this project[7].

4.2.1 Infrastructure provisioning

The GitLab project contains a Terraform folder which stores all the code files necessary to build the infrastructure. These files are the following:

1. *main.tf*: Main file that uses others to build provision the infrastructure and output results. It is this file that sticks a floating IP to each virtual machine.

2. *network.tf*: Defines the virtual network, router, and subnet configuration. It also gives DNS nameservers to virtual machines so they can browse the web like normal computers (and access mirror repositories to download packages...).

3. *providers.tf*: Defines the providers used by Terraform. There is only OpenStack.

4. *security_group.tf*: Defines how the virtual router (which also acts like a firewall) should filter packets with a virtual machine as destination, of as sender. To avoid any problem, and because this project is made in a **development environment not accessible from Internet**, most of the ports have been opened.

5. *ssh_key.tf*: Defines the SSH key automatically added to the virtual machines. A specific one has been created for Ansible, so that the tool can directly configure the machines without any required action from the administrator.

6. *version.tf*: Defines the minimum version required for the OpenStack provider. Here, the version must be equal or over of 0.14.0.

7. *variables.tf*: Defines all variables of the infrastructure, including the number of virtual machines, their name, operating system, flavor, network attachment...

These are only "pure" Terraform files used to provision the infrastructure. In addition of these, there are other files that are used to automatically create configuration files used by Ansible and by the benchmark itself:

1. *ansible-hosts.tpl*: Template file filled by Terraform. The final result is the inventory file used by Ansible to access to all virtual machines.

2. *mpi-hosts.tpl*: Template file filled by Terraform. The final result is the hosts file used by OpenMPI to access to all virtual machines.

This automation process is very interesting, and some more explanations are available in the appendix B.

A simple example could be that an administrator wants a cluster of 8 virtual machines, deployed on the green OpenStack. He just needs to pull the GitLab repository, open the *variables.tf* file and change the following part of the file to whatever he wants:

```
variable "vm_names" {
  description = "VM Names"
  default = ["HPL01","HPL02","HPL03","HPL04","HPL05","HPL06"]
  type = set(string)
}
```

He is also able to change the flavor, operating system and other parameters depending on these available on the OpenStack cluster. Once all variables are set, the plan can be executed to check everything, and then applied to validate the provisioning of the new infrastructure.

4.2.2 Virtual machine configuration

This section now also concerns the VMware cluster, as the virtual machines are provisioned and empty of any configuration.

The VMware virtual machines needs to be configured with some commands to enable the Ansible access. This approach is described in the appendix C.

The GitLab repository contains an Ansible folder with all configuration, inventories, playbooks and others files needed to fully automate the benchmark installation and execution. The OpenStack inventories files are automatically created by Terraform. Note that the inventory of the VMware cluster has been created manually.

Benchmark installation

The benchmark is installed on each virtual machine like it has been done on the modern laptop, but this time using Ansible playbooks. Each component of the benchmark has its own playbook, that can be replayed to change a version, fix a bug, or whatever without having to redeploy the full stack. All these playbooks are combined in one upper playbook, called *all-in-one.yml*. This playbook only has 6 lines:

```
---
# Approximately 15 minutes
- import_playbook: ./includes/pre-config.yml
- import_playbook: ./includes/install-mpi.yml
- import_playbook: ./includes/install-blas.yml
- import_playbook: ./includes/install-hpl.yml
```

All it does is executing each playbook one after the other. The *pre-config.yml* playbook applies some basic configuration to virtual machines, and installs requirements like Fortran and C compilers and other tools. This section will not describe each playbook as it is the same procedure as explained above, in the first section of this chapter. The GitLab repository also stores configuration files like the *Make.linux64* used by the HPL compiler process, and the OpenMPI hosts files generated by Terraform. These files are pushed on the virtual machines when needed. The approximate time to install the benchmark stack on each virtual machine is 15 to 20 minutes.

Executing the benchmark

The benchmark execution is managed in the *run-full-hpl.yml* playbook file. It looks like this:

```
---
- name: HPL execution playbook
  hosts: hplmaster
  gather_facts: yes
  become: false
  vars:
    hpl_n: 28800 # matrix size
    hpl_p: 1 # HPL P
    mpi_core_per_node: 8 # number of cores per node

  tasks:
  - set_fact:
        np_iteration: # HPL Q and MPI -np parameter
          - 1
          - 2
          - 3

  - name: Start HPL for multiple configuration
    include_tasks: ./includes/run-hpl-three-times.yml
    loop: "{{ np_iteration }}"
    loop_control:
      loop_var: np_var
```

The playbook is executed on a specific group of the inventory, called *hplmaster*. There is only one virtual machine in this group. Terraform puts in it the first virtual machine defined in the variable *vm_names*. This is done because only one virtual machine needs to start the benchmark, and push the problem to other hosts. If this playbook was applied to all virtual machines, each machine would have to perform the benchmark x times concurrently, with x the number of virtual machines.

The playbook file contains an iteration variable, defining the number of cores to be allocated to the problem. As this project sets P parameter to 1, these values are also equal to Q parameter (and *-np* OpenMPI parameter). For each value set by the administrator in the *np_iteration* variable, the playbook will call another playbook called *run-hpl-three-times* that will run the benchmark 3 times, with each configuration. In this way, the results presented represents the average of the 3 tests and not a single value. In the example above, the benchmark will be run 3 times with Q parameter set to 1, then 3 times with it set to 2, and finally 3 times with it set to 3. The values of the *np_iteration* don't necessarily have to be continuous, it could be 1, 4, 6, 9... Or even 9, 3, 2, 1.

Each benchmark result is stored in a file called *hpl-[np value]-[iteration]*. For example, the second time the benchmark has been executed with the *-np* OpenMPI parameter set to 5, the output file name is *hpl-5-2*. This way, all results are stored in different files to keep them across time with an explicit name. At the end of each benchmark execution (when an output file is complete), the file is automatically pulled on the Ansible control node (most of the time a laptop that started the *run-full-hpl.yml* playbook). All the results are stored in a folder called *results* inside the Ansible folder, but this one is git ignored.

4.3 Containerization and deployment on Kubernetes

This section explains how the benchmark was containerized and the various stages involved in deploying it on Kubernetes.

4.3.1 Build of the Docker image

During the project, and with the deadlines looming, it was decided to go for the simplest way of containerizing the benchmark, maybe by increasing the project's technical debt[13]. As a result, there was no CI/CD pipeline creation and no auto-build of the Docker image for each push. This part represents a typical point of improvement in the project, although the Docker image finally created is perfectly usable.

The idea to containerize the benchmark is simple: start from the inside of an Ubuntu 22.04 LTS image, install the whole benchmark stack, and save a new version of the image with the command *docker commit*. 2 images have been created because of the different architectures used in this project. For testing purposes some containers have been executed on the MacBook Pro M2, using an Apple Silicon chip based on an ARM architecture. The Docker image committed on the laptop was made for an ARM-only machine. To build the image for an AMD architecture, one virtual machine emulated on the OpenStack cluster has been used. The AMD image is stored on the Docker Hub, in the *leichap/self-hpl-3* repository. It can be pulled publicly, without any credentials. This way, and because this is not considered as a private image, it can be used by other people to perform HPL benchmarking in containerization environments.

4.3.2 Kubernetes project

To deploy the project from an empty Kubernetes project, only 2 files have been needed. In fact, this project does not focus on the Kubernetes configuration, as the pods must be up and running only during the benchmark without any specific configuration. The GitLab repository stores the 2 configuration files. One for the namespace creation, and the other for the deployment. Note that the deployment file is simple:

```
---
kind: Deployment
apiVersion: apps/v1
metadata:
  name: hpl
  namespace: hpl
spec:
  replicas: 2
  selector:
    matchLabels:
      app: hpl
      version: v01
  template:
    metadata:
      labels:
        app: hpl
        version: v01
    spec:
      containers:
      - name: hpl
        image: leichap/self-hpl-3:latest
        imagePullPolicy: IfNotPresent
```

This is a general deployment file, without any specific configuration like resources limits or PodAffinity rules. These rules were specified in the iCoSys Rancher GUI, and they define that each pod must be on a different worker node, excluding GPUs nodes.

Once the pods are up and running, it is needed to access to one to start the benchmark. All pods can access each other by their private IP address, but at least one must be accessible by the administrator to configure and start the benchmark.

It would have been possible to automate this process by exposing a pod with a public IP address, using a NodePort configuration or a DNS-Service. Ansible would then have access to the pod and would be able to perform same actions as on virtualized environments. Due to lack of time, this has not been done.

To access a pod, the Rancher GUI has been used. From here, the administrator must fill the hosts file used by OpenMPI with each pod's private IP address and the number of cores allocated to each pod. It has been decided to follow the same procedures as for the virtual machines, using 8 cores per node. The benchmark is installed at */root/hpl-2.3/bin/linux64/* folder. The benchmark can then be started using the following commands:

```
/root/openmpi/bin/mpirun -np [Q parameter value] --prefix /root/openmpi
↪  --allow-run-as-root -hostfile mpihosts xhpl
```

The fact that the OpenMPI *-np* parameter must be equal to Q HPL parameter is only true in this project, because it has been choosen to fix P parameter to 1. Note that OpenMPI can rise an exception saying that it is discouraged to run an OpenMPI program as root, but this is handled by adding the *–allow-run-as-root* flag. Because this is done in containers, only executed for testing and benchmarking purposes, this is not considered as an issue.

At the end of the benchmark, the result will be available on screen. If it is required to save the output in a file, the command can be piped in a file without any problem.

5 Results and interpretation

This chapter discusses the results obtained at the end of all the tests, and presents the conclusions drawn. It is important to remember that all the experiments have been done in a strong scaling procedure. The appendix A shows the repartition of the xhpl processes among all hosts.

The first part of this chapter shows the raw results obtained. These results are compared to the result of the experiment on a modern laptop, and then the following sections tries to answer this question: "What cluster seems to be best-suited for HPC ?". At the end of this chapter, some other experiments made during the project are presented, sometimes without any conclusion because the experiments are not finished. A list of experiments ideas is also presented.

Contents

5.1 Raw results

This part aims to show the results as they have been collected, without any special treatment.

The results of the benchmark can be viewed on the following graph:

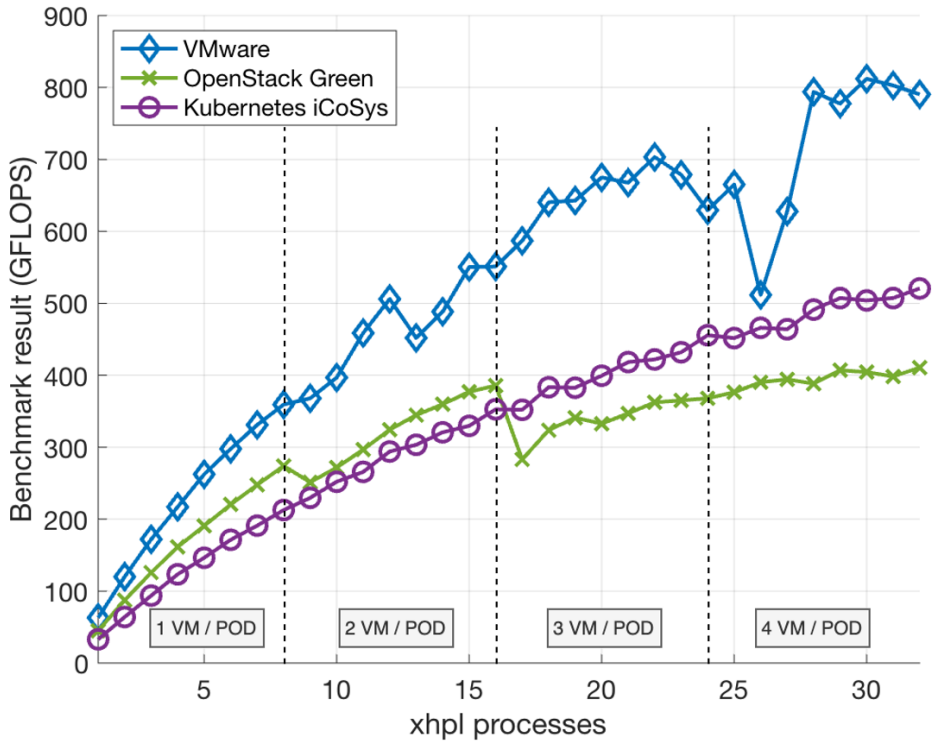


Figure 5.1: Raw results of the HPL benchmark executed on 3 different clusters

We can see that the VMware cluster has better performance than the OpenStack and Kubernetes. It is also clear that the VMware curve is not linear, where the Kubernetes growth is constant. On the OpenStack cluster, the performance losses when a new virtual machine is added to the benchmark is clearly highlighted.

The big loss on the VMware cluster (when xhpl process is equal to 26) should not be considered as a measurement error, as this test case has been executed 12 times, and the value shown on the graph is the average. The Ansible playbook runs the benchmark only 3 times for each test case, but the playbook has been re-executed multiple times to obtain a more accurate average.

These suddent jumps in the blue and green curves can be explained by the fact that the network limits the benchmark's performance. For the VMware, one other explanation can be that the cluster is used by a lot of people (573 VM are emulated on it, told

François Buntschu on 01.23.2024). As more people are using the cluster, more the network usage is high and the bandwidth is shared between all virtual machines, same for the physical CPUs of blades. It is possible that the benchmark has been executed during a high-usage time of the cluster.

The losses view on the OpenStack cluster highlights the bottleneck that represents the network. When the benchmark is executed on a single virtual machine, the bandwidth available between cores is approximately 15 Gbps (value took from an iperf3 measurement). When multiple virtual machines must communicate, they use the network of the hypervisor, which has a bandwidth of 3 Gbps (iperf3). The fact that the network is fully used during the benchmark makes the xhpl processes waiting for resources. During these waiting times, they are not calculating anything and are just doing busy waiting to keep the execution context of the process sticked to the core.

Note that the Kubernetes cluster does not looks like it is impacted by the network usage when using multiple pods, even is they are executed on different worker nodes.

An other way to highligh the losses due to the network is to show the variation on the graph, which is done below:

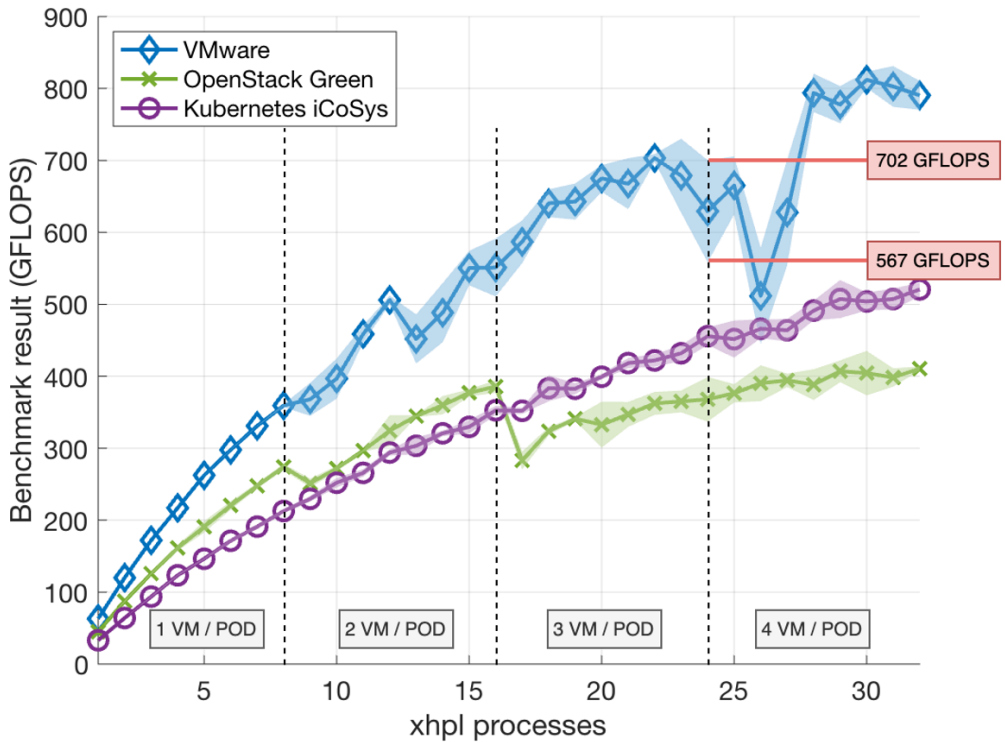


Figure 5.2: Variation shown for each test case

When the benchmark is executed on a single virtual machine, with an almost infinite bandwidth between cores, the variance is nearly null. As the CPU are almost never waiting for resources, they can calculate all the time, and the results are very similar.

When the hypervisor network is involved, the benchmark result directly depends on the quality and speed of transmission. More the CPU are waiting for resources, worst the benchmark result will be as the number of FLOPs required to calculate the problem will be executed in more time. As an example, the minimum and maximum values are shown on the graph for the test case when there are 24 xhpl processes launched, on the VMware cluster.

5.2 Comparison to modern laptop

One objective of this project is to compare the computing capacities of different clusters with a modern laptop. A MacBook Pro M2 has been used, with the same benchmark parameters used for the clusters.

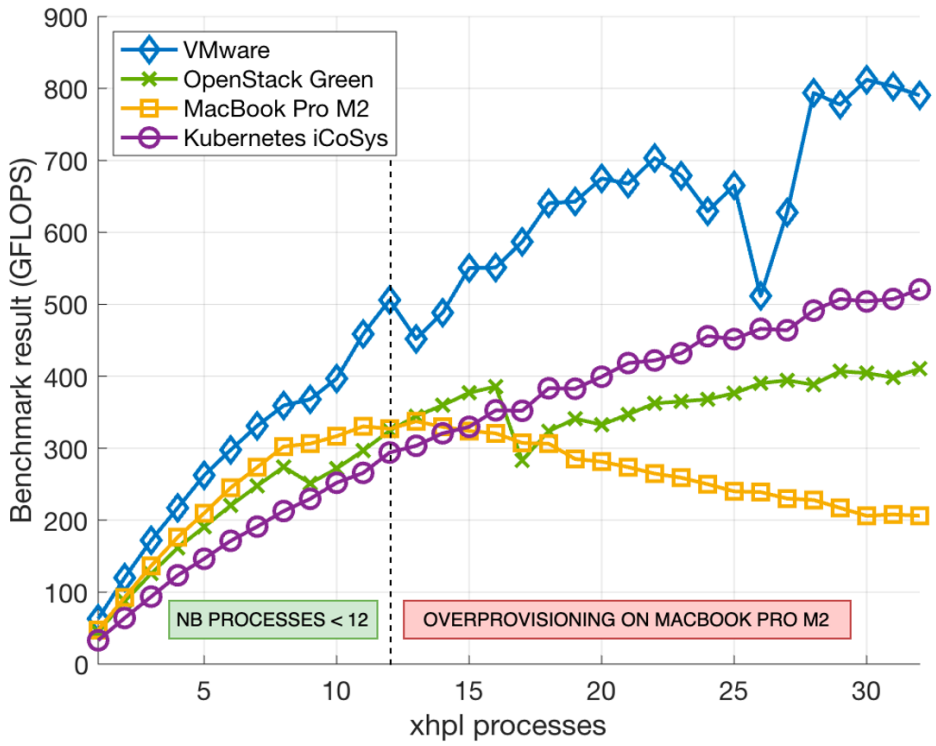


Figure 5.3: MacBook Pro M2 compared to ISC clusters

We can clearly see that the MacBook curve has a different trend to the others, as if there was a point where the best performance was achieved, and that all the following test cases are worse.

The MacBook Pro M2 used on this benchmark has only 12 cores, divided in 2 categories:

- 8 "high-performance" cores.
- 4 "high-efficiency" cores.

The curve seems to be growing smoothly on the first 8 cases, but when the benchmark is also executed on some efficiency cores (from 9 to 12), the curve is growing more slowly. The high-efficiency cores are made to find a balance between performance and electrical consumption, where the high-performance cores are made to obtain the best performance.

The situation where the number of xhpl processes is over the number of cores allocated to the benchmark is call an overprovisioning situation. The computer must switch between different execution context to ensure that all processes have the same CPU time allocated. The more processes are launched on the machine, the more the cores must change of contexts, so they have less time to calculate the initial problem.

When the MacBook Pro M2 is not in an overprovisioning situation, its computing performance is perfectly acceptable compared with clusters. For the rest of this document, the MacBook Pro M2 is not shown on graphs as the overprovisioning situation does not reflect the real computing capacity of the machine.

5.3 What cluster seems to be best-suited for HPC

A hasty conclusion might be to consider the VMware cluster as the best cluster for distributed computing, as it has the best performance compared to other clusters, for equal resources.

Actually, it depends on the definition of "best cluster for HPC". If the only factor is the raw result, then the VMware cluster can be considered as the best for HPC. If other factors are considered, like variation, performance relative to a single process, or the CPU utilisation efficiency, the ranking may be different.

5.3.1 Scaling efficiency

The normalized graph is equivalent to show the "scaling efficiency coefficient", or the "relative performance to a single process" of each cluster. Each point of the curve is divided by the first point of it. A perfect case would be when the scaling efficiency coefficient follows the equation $y = x$. This would mean that when adding xhpl processes to the problem, the time needed to compute is fully proportional by the number of processes. The number of FLOPS would then be the FLOPS obtained when using one process multiplied by the number of processes allocated to the problem.

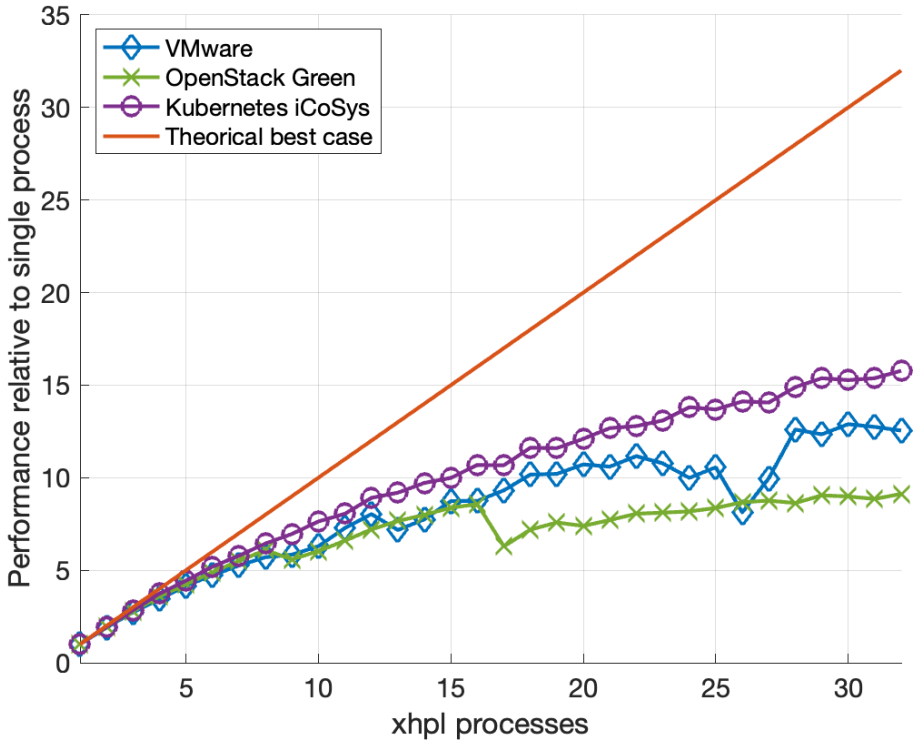


Figure 5.4: Normalized graph, showing the scaling efficiency coefficient for each cluster

On this graph, the VMware is not considered as the best cluster for HPC. The iCoSys Kubernetes cluster seems to have a better scaling efficiency than the VMware, making it more suitable for HPC tasks.

In fact, is it not possible to follow the $y = x$ equation. All the curves show the same tendency to deviate from the theoretical curve, and it is certainly due to the overhead implied by distributed computing. The overhead is all the additional costs incurred in managing the parallelization and distribution of the problem, compared to performing the benchmark on a single processor. The inter-process communication using MPI and the IP network can be included in the overhead. It is not exclude that some other reasons can explain the trend of the curves.

The normalized graph can also be used to compare the behaviour of the clusters to a theoretical law, called Amdahl's law (see below, section 5.3.3).

5.3.2 CPU usage efficiency

An other way to decide which cluster is the best-suited for HPC tasks is to compare the CPU usage efficiency, in other words what is the proportion of time during which the cores are actually processing the problem, compared with the waiting times associated with the overhead (network communication, waiting for resources...). The theorical line shows a case where the CPU are always computing the problem, without any interruption due to communication process.

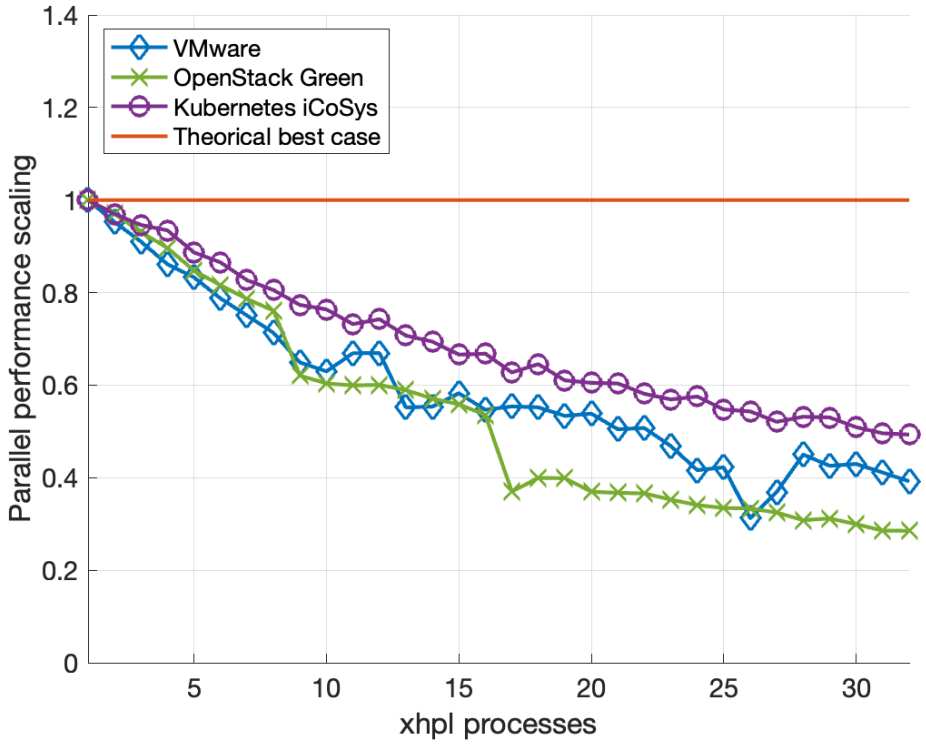


Figure 5.5: CPU usage efficiency

Once again, the Kubernetes cluster seems to be the best. It is possible to say that when there are 32 xhpl processes running conccurently, the cores are really processing the problem 50% of the time. It means that the pods running on Kubernetes are less waiting for resources than on the other clusters. This can be explained by the fact that the Kubernetes network bandwidth is better than the other clusters or simply that the Kubernetes internal network is using better network equipment (10 Gbps switches...), or is perfectly configured. If a NIC on a host or network device is misconfigured, the bandwidth can be affected. Here are the comparison of the bandwidths:

- Kubernetes cluster: **6.75 Gbps**
- VMware cluster: Not applicable as at the end of the project, the 4 virtual machines are on the same blade (internal bandwidth calculated: **33.5 Gbps**).
- OpenStack green cluster: **3.5 Gbps** (possible identified network problem, as it should be around 10 Gbps)

Note that when performing the experiments on the VMware cluster, the virtual machines were possibly on different blades. Since then, new blades have replaced old ones and it is impossible to retrieve the bandwidth between old blades.

5.3.3 Regression of Amdahl's law on normalized graphs

In 1967, a computer scientist named Gene Amdahl published a law that took he's name, the Amdahl's law. He explained that the speedup of a program using multiple processors in parallel computing is limited by the proportion of the program that can be parallelized. For example, if only 50% of a program can be parallelized, the maximum theoretical speedup using parallel computing would be 2, regardless of how many processors are used. This law emphasizes the importance of identifying and optimizing the serial portion of a program to achieve significant improvements in overall performance when using parallel computing. This law can be shown as a graph, like this:

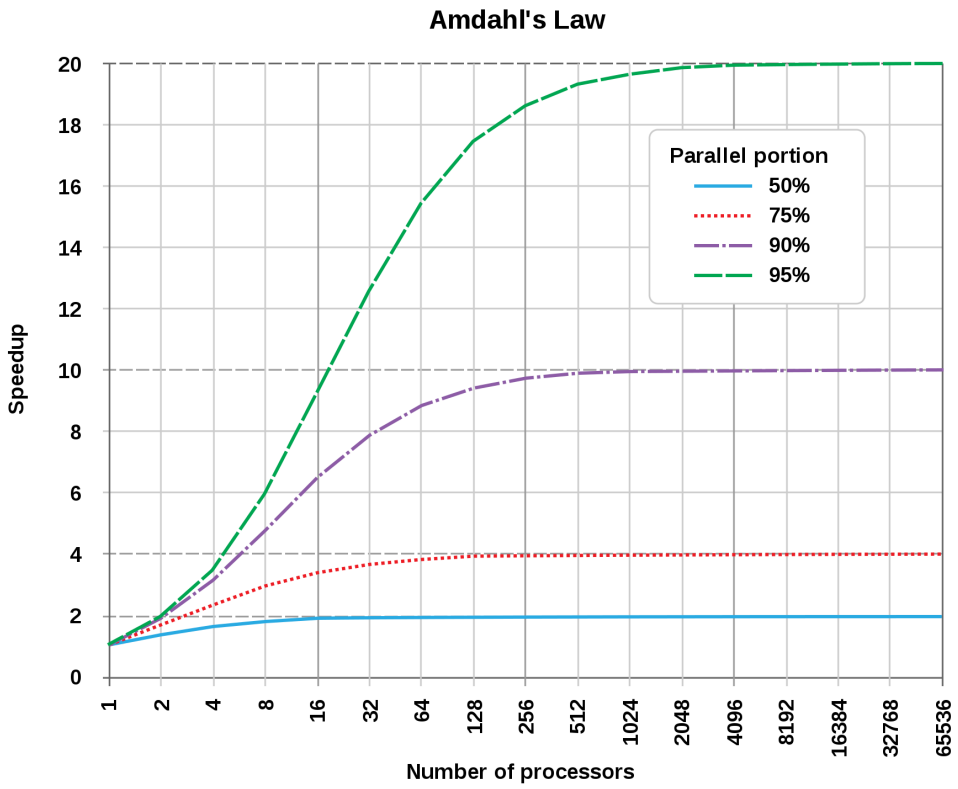
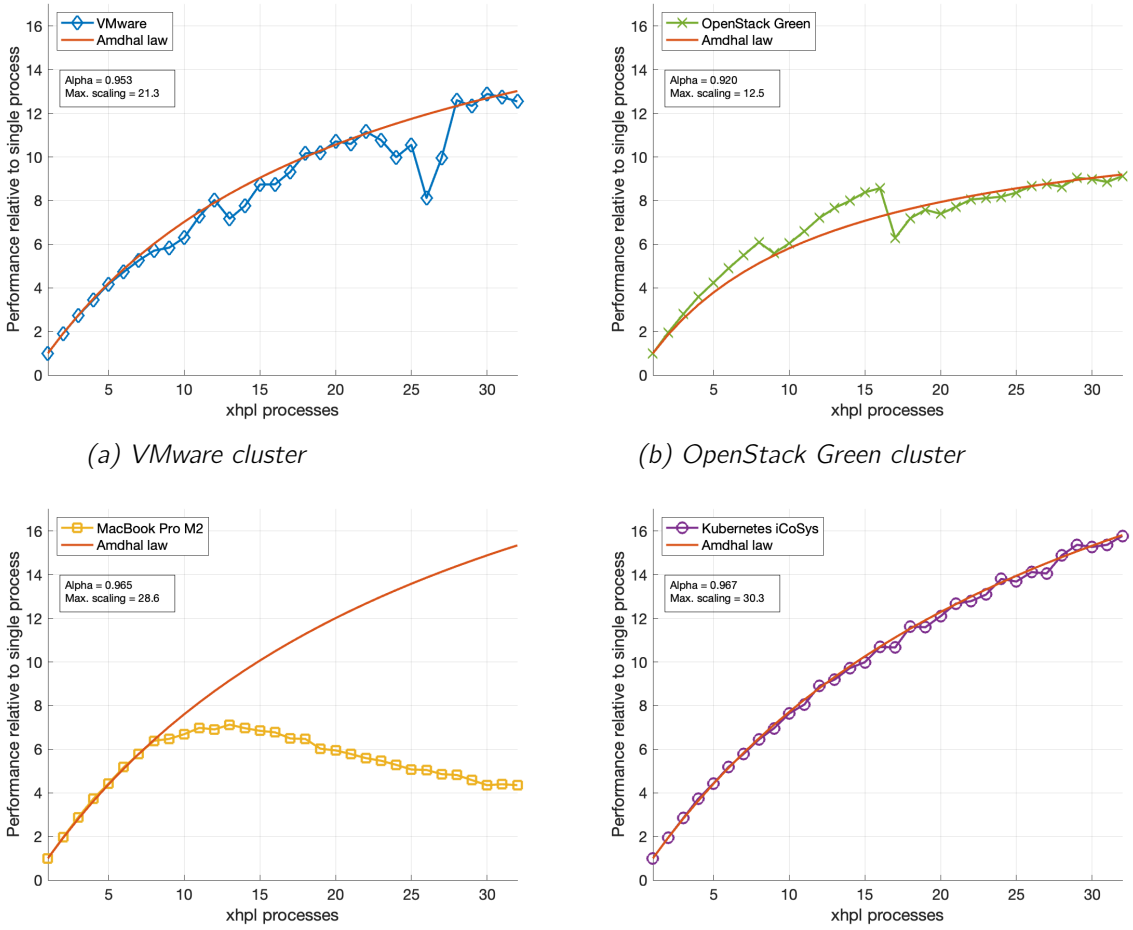


Figure 5.6: View of Amdahl's law (Wikipedia)

The parallel portion of a program is called the α parameter. If $\alpha = 0.5$ then the parallel portion of the program is 50%. It is then possible, from the normalized graphs obtained for each cluster, to find an α value that fits the normalized curve. With this, it is possible to estimate the maximal theoretical speedup that each cluster can reach with an infinite of cores allocated to compute the problem.



(a) VMware cluster

(b) OpenStack Green cluster

(c) MacBook Pro M2

(d) Kubernetes cluster

Figure 5.7: Regression of Amdahl's law on each curve

For each cluster, the graph shows the maximal scaling (speedup) that it could reach with an infinity of xhpl processes and cores. The α parameter can be multiplied by 100 to obtain the parallel portion of the problem. The VMware cluster curve shows a maximal speedup of 21x, where the OpenStack is only at 12x. Note that for the VMware, OpenStack and Kubernetes clusters, the red line shows the scaling taking into account the network losses, this is why some test cases are over the curve. The α parameter has been set to fit the last data point. On the other side, on the MacBook Pro M2, the curve has been drawn to "imagine" what the result could have been if the machine was not in an overprovisioning situation. This is not reflecting the real maximal speedup of the laptop, as it should be much lower. This has still been done to show that the Amdahl law can be used to extrapolate results from a given data set.

The Kubernetes cluster is showing a maximal speedup of 30, which is the best result between all clusters. It is important to note that the Amdahl's law is idealistic, as it does not take into account the overhead implied by the parallel computing. This is why the curves obtained does not follow a perfect Amdahl curve, except for the Kubernetes

cluster that shows an impressive continuity. There is no precise explanation of why the Kubernetes cluster does such a great job, it can be due to a better internal network bandwidth, usage, efficiency... More investigations are needed to find out.

5.4 Other experiments

This section explains other experiments that were carried out during the the project, but have not been completed or are not fully explained for other reasons. The conclusions drawn in this section may be imprecise, and there is potentially a lot of work left to continue the experiments.

5.4.1 VMware blades comparison

An other experience has been made during this project when the VMware cluster blades have been changed. 2 old blades were replaced by 2 new, and the benchmark has been executed before and after this changement.

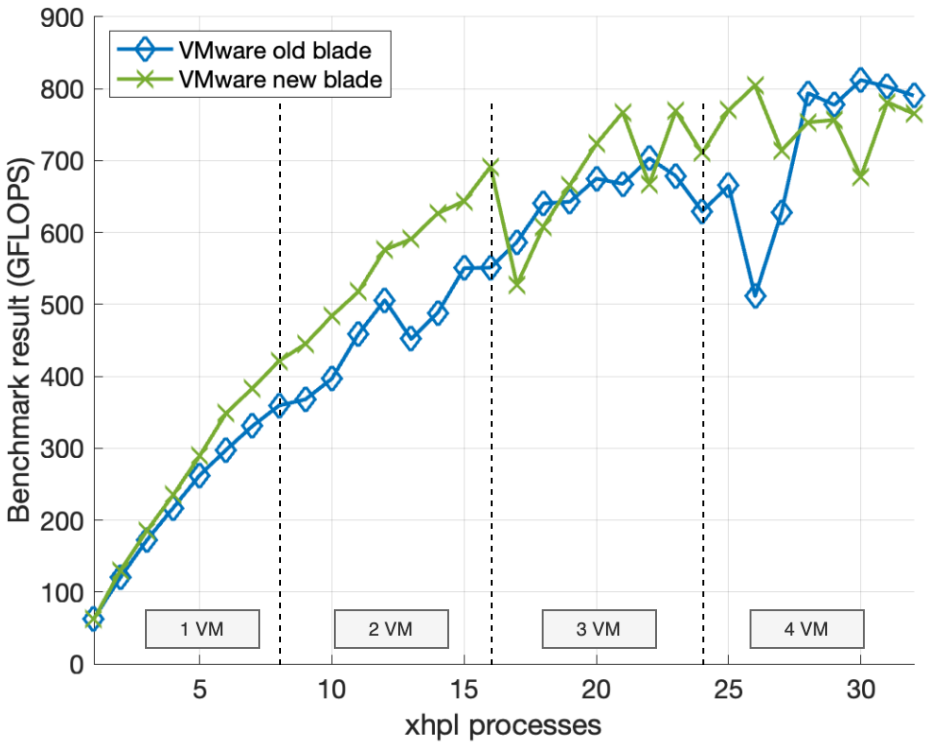


Figure 5.8: Comparison of the old and new blades of the VMware cluster

It it possible to see that when performing the benchmark on one virtual machine, the new blade is offering better performances. This is also true with 2 virtual machines, but when a third and a fourth one are added, the results are less convincing. There has been no further study of this test case. It is possible that the network is one of the

main causes of this result, without any insurance. It is also possible that, because the 4 virtual machines are running on the same new blade, the physical server must swap between virtual machines.

5.4.2 Electrical consumption

Some work has been done to monitor the electrical consumption of blades while computing the benchmark. The experience have been monitored on the OpenStack and Kubernetes cluster, and also on the MacBook Pro M2. It has not been possible to access to the values of the VMware cluster due to a lack of permissions. The OpenStack and Kubernetes cluster were monitored using the IPMI interface of blades, and displayed on a LYSR[14] dashboard. For the MacBook Pro M2, a tool prints on the screen the total power consumed by the laptop. The result, given in GFLOP/Joule shows how many GFLOP can be computed using 1 Joule. More is better.

OpenStack cluster
Taking the last result of the cluster, when having 32 xhpl processes, the number of GFLOPS is equal to 400. The LYSR dashboard shows 300 Watts per blade, which means that the total consumption is $300 * 2 = 600$ Watts, and then:

$\frac{474}{600} = 0.79$ GFLOP/Joule

Kubernetes cluster
Taking the last result of the cluster, when having 32 xhpl processes, the number of GFLOPS is equal to 520. The LYSR dashboard shows 160 Watts per blade, which means that the total consumption is $160 * 4 = 640$ Watts, and then:

$\frac{520}{640} = 0.81$ GFLOP/Joule

MacBook Pro M2
Taking the result when having 12 xhpl processes, the number of GFLOPS is equal to 325. The MacBook Pro M2 indicates a consumption of 70 Watts during the benchmark. Then:

$\frac{325}{70} = 4.64$ GFLOP/Joule

5.4.3 More on the network losses

Here are a few experiments carried out at the end of the project to find out more about the losses supposedly due to the network.

Comparing floating IPs to private IPs
This experiment has been made to see the difference between the usage of the "public" network and the internal network of the hypervisor. Theoretically, the internal network should deliver better performance because there are less network devices. The floating IPs are sticked to the virtual machines, but the traffic must go out and re-enter by the virtual router of the private network.

Cluster involved in this experiment: OpenStack Green

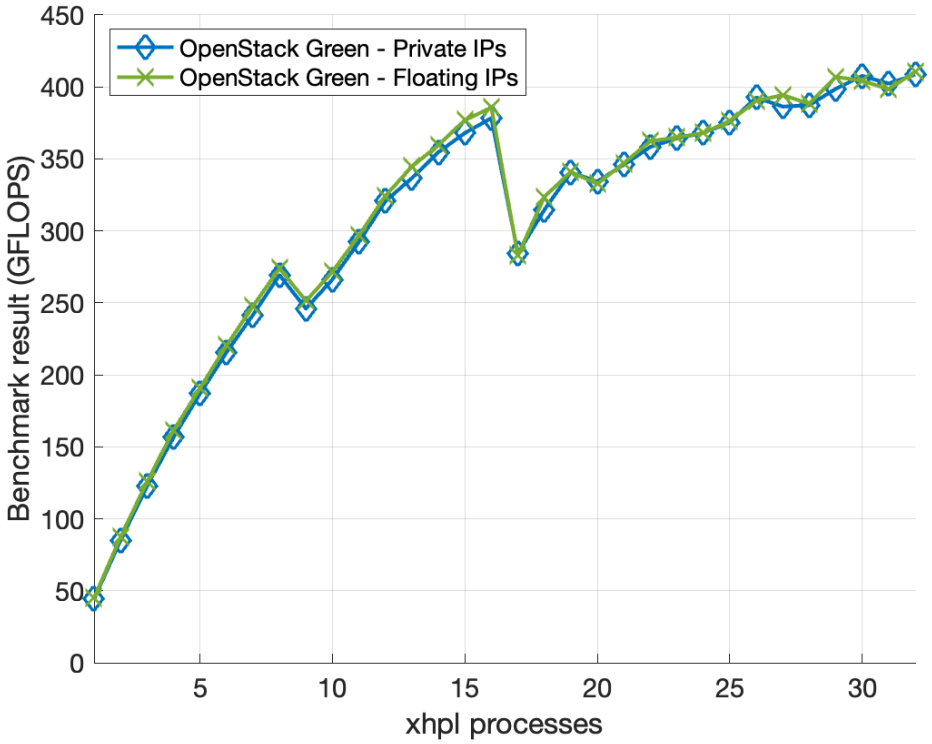


Figure 5.9: Private IPs versus floating IPs on the OpenStack green cluster

It is possible to see that there is no significant difference between the use of private and public IPs. It all depends on the bandwidth available for each path. These bandwidth are given below (using iperf3) with 2 virtual machines on different blades:

- Bandwidth using private IPs: **4.5Gbps**
- Bandwidth using floating IPs: **3.5Gbps**

A difference of 1Gbps in the bandwidth seems to not affect a lot the benchmark results, but more investigations are required to evaluate the real differences. This experiment was not carried out on VMware, because the virtual machines only have public IPs, or on Kubernetes, because deploying public IPs for pods requires more configuration and time. It is also strange that the private network's bandwidth is 4.5Gbps, it is possible that some network devices or NIC are misconfigured and does not allow a higher bit rate.

Comparing the internal network of a blade and the hypervisor network
This experience aims to show the difference between running 2 virtual machines on the same blade and on different ones. When running both on the same blade, they can certainly use the blade's internal network offering a better bit rate.

Cluster involved in this experiment: OpenStack Green

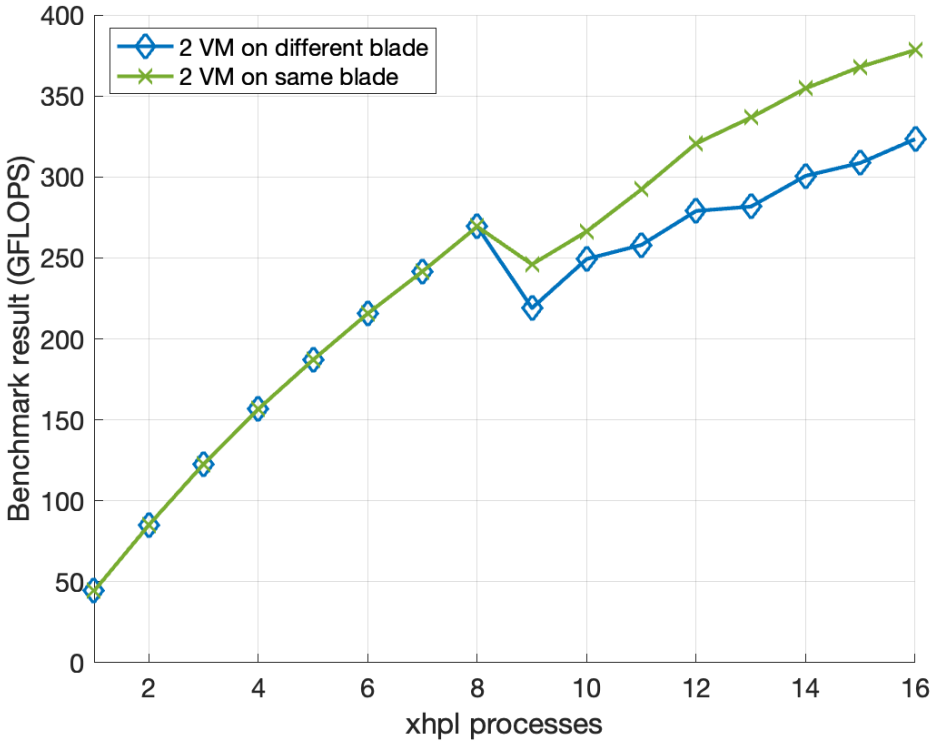


Figure 5.10: Comparison of 2 virtual machines on the same blade, and on different ones

On the first 8 test cases, the results are the same because only one virtual machine is computing the problem. When a second one is added, it looks like the performance is better when the virtual machines are running on the same blade, confirming the hypothesis that the internal of the blade network is better than the one linking all blades.

5.4.4 Ideas of experiments

Here are presented some ideas of experiments. They have not been executed by lack of time, permissions, or because they were out of the scope of the project.

- Performing the benchmark between the different clusters. It would be interesting to see the addition of each cluster when working together. Having 4 virtual machines running on each cluster, plus 4 pods on the Kubernetes cluster would give a total of 12 hosts. The result should not simply be "(GFLOPS of OpenStack) + (GFLOPS of VMware) + (GFLOPS of Kubernetes)" because the communication process between all hosts will be longer than usual. Also, the number of cores allocated to the problem will be too high for the problem size: the hosts will be waiting for resources too much time.

- Performing the benchmark using different network bandwidths. For example, with a 1Mbps link between 2 virtual machines, then 10Mbps, 100Mbps, 1Gbps, 10Gbps...

- An other way to see how the clusters behave is to make the problem go bigger, while not changing the number of processes. The X axis will be the problem size and the Y axis will still be the benchmark result.

- Monitor the bit rate of the benchmark for each cluster and test case to give results in Bytes/FLOP representing the number of bytes sent on the network to perform 1 floating point operation.

- The iCoSys Kubernetes cluster is equipped with GPU worker nodes, and they have been excluded of this project because the purpose was to compare the computing power of each cluster using their CPUs. Anyway, the comparison between CPU-bases worker nodes and GPU-bases worker nodes could be interesting.

- Performing the benchmark by adding an xhpl process with a round-robin method on each virtual machine. The network losses should be visible from the first cases, but the global curve may be smoother.

6 | Conclusion

This chapter is the final one of this report, giving the conclusions of the project, the comparison with the initial objectives, the perspectives and a personal conclusion.

The initial objectives of this project were achieved, as the chapter 5 shows the measures and comparisons of the computing power of the multiple clusters. This chapter also indicates how the clusters behave when being under stress, by analyzing their electrical consumption and results. The question "What cluster seems to be best-suited for HPC" has also been answered, depending on the definition of "best-suited for HPC". If this means that the only determinating factor is the result of the benchmark, the VMware cluster is the best, followed by the Kubernetes cluster and the green OpenStack. If some other factors are taken into account, like the scaling efficiency, CPU usage efficiency and best theorical speedup given by Amdahl's law, then the Kubernetes cluster is the best, followed by the VMware and OpenStack. It's also impressive to see how the Kubernetes cluster behaves, with very regular curves.

It is important to note that the HPL benchmark only gives a partial view of the answer. Indeed, the tasks performed by HPL only reflects a specific kind of computation. A lot of other benchmarks exists and reflects other kind of workloads. For example, a machine learning oriented benchmark would be a good way to compare the clusters from a practical view. Anyway, to obtain a full comparison between multiple clusters it would be required to perform multiple kinds of benchmarks, and compare the different final results. It is also important to point out that the aim of this project is to indicate which cluster seems best suited to distributed computing, without fully explaining why. Some partial answers were given in the chapter 5 but a lot of experiments are still incomplete.

Finally, the HPL benchmark is a good way to compare different clusters between them, but there are a lot of factors to take into account before performing the test. For example, there is always the question of which blade the virtual machines are located on, whether the situations between the clusters are perfectly equitable, whether traffic and virtual machines that do not concern this project can impact the results, etc. It also depends a lot on the BLAS implementation and the benchmark configuration. I'm sure there's still a lot to be done to improve the testing procedure of this project and to make more observations on the behaviour and comparison of HEIA-FR clusters.

I really enjoyed working on this project, which combined aspects of project management, automation, networking, distributed computing and research. I got a taste for trying to automate as many things as possible to save time so that I could do more experiments, and today I'm proud to see the different situations I've been able to test. I hadn't touched on the subject of distributed computing before and I'm now aware of the difficulty of the task and what's at stake. I sincerely hope that this report reflects the pleasure I had in carrying out this work, from the very first day until this document was sent out.

Martin Roch-Neirey

A Distribution of xhpl processes across nodes

Node 1	Node 2	Node 3	Node 4	xhpl processes count
1	0	0	0	1
2	0	0	0	2
3	0	0	0	3
4	0	0	0	4
5	0	0	0	5
6	0	0	0	6
7	0	0	0	7
8	0	0	0	8
8	1	0	0	9
8	2	0	0	10
8	3	0	0	11
8	4	0	0	12
8	5	0	0	13
8	6	0	0	14
8	7	0	0	15
8	8	0	0	16
8	8	1	0	17
8	8	2	0	18
8	8	3	0	19
8	8	4	0	20
8	8	5	0	21
8	8	6	0	22
8	8	7	0	23
8	8	8	0	24
8	8	8	1	25
8	8	8	2	26
8	8	8	3	27
8	8	8	4	28
8	8	8	5	29
8	8	8	6	30
8	8	8	7	31
8	8	8	8	32

Table A.1: Distribution of xhpl processes across nodes

B | Use Terraform to create local files

The Terraform project contains 2 extra-files used to populate Ansible inventories and OpenMPI:

1. *ansible-hosts.tpl*: The final result is the inventory file used by Ansible to access to all virtual machines. It looks like this:

```
# File generated by Terraform, content modified by user may be lost
# Martin Roch-Neirey, 2023

[hpl]
%{ for ip in floating_ips ~}
${ip}
%{ endfor ~}

[hplmaster]
${floating_ips[0]}

[hpl:vars]
ansible_user="debian"
mpi_hosts_file="${cluster}-mpi-hosts"
ansible_ssh_common_args="-o StrictHostKeyChecking=no"
ansible_ssh_private_key_file=/Users/martinrn/.ssh/ansible_key
```

   When the Terraform process is nearly ended, the infrastructure is provisioned and each virtual machine has a floating IP attached to it. Terraform has the list of the floating IPs used, and simply writes it in the file. For the *hplmaster* Ansible group, Terraform only writes the first IP of the list, so that only one virtual machine is in this group. Some other Ansible variables are written below to ensure that virtual machines can access each other using SSH.

2. *mpi-hosts.tpl*: The final result is the hosts file used by OpenMPI to access to all virtual machines. It looks like this:

```
# File generated by Terraform, content modified by user may be lost
# Martin Roch-Neirey, 2023

%{ for ip in floating_ips ~}
${ip} slots=SLOTS_VALUE
%{ endfor ~}
```

   In this file, Terraform also writes each floating IP, and adds at the end of the line "slots=SLOTS_VALUE". This is done so that Ansible can then replace "SLOTS_VALUE" by the number of cores available on each virtual machine, when starting a new benchmark task.

In the *main.tf* Terraform file, these 2 templates files are referenced like this:

```
# Get floating IPs from OpenStack
locals {
  floating_ips = [for ip in openstack_networking_floatingip_v2.hpl : ip.address]
}

# Generate Ansible inventory
resource "local_file" "green_ansible_inventory" {
  count   = var.cluster == "greenOS" ? 1 : 0
  content = templatefile("${path.module}/ansible-hosts.tpl",
    {
      floating_ips = local.floating_ips,
      cluster = var.cluster
    }
  )
  filename = "../ansible/inventories/greenOS-hosts"
}

# Generate OpenMPI hosts file
resource "local_file" "green_mpi_inventory" {
  count   = var.cluster == "greenOS" ? 1 : 0
  content = templatefile("${path.module}/mpi-hosts.tpl",
    {
      floating_ips = local.floating_ips,
      cluster = var.cluster
    }
  )
  filename = "../ansible/playbooks/files/greenOS-mpi-hosts"
}
```

The floating IPs are stored in a variable called *floating_ips*, and then this variable is used as a list to fill the files. More information on the *local_file* principle can be found on Terraform official documentation[15].

C | Prepare a new VMware VM for the benchmark

In order to use a virtual machine emulated on the ISC cluster as an MPI host, it is necessary to perform some preparation tasks.

The following procedure must only be considered when using a VMware virtual machine, provisioned on the ISC cluster. For the example, the following virtual machine is used :

- Empty of any user-side configuration.
- Ubuntu 22.04 LTS
- Accessible via SSH with a 8.8 username and AAI password.

Ansible must have access to the virutal machine to install the HPL stack, start the benchmark and gather results.

1. Add Ansible public key in this file:

```
sudo /home/ubuntu/.ssh/authorized_keys
```

2. Enable public key authentication in the SSH server configuration file:

```
sudo /etc/ssh/sshd_config
PubkeyAuthentication yes
AuthorizedKeysFile .ssh/authorized_keys
```

3. Authorize "ubuntu" user to elevate it's privileges:

```
sudo visudo
```

   Add at the very end of the file:

```
ubuntu ALL=(ALL) NOPASSWD:ALL
```

4. Be careful that in your Ansible playbooks and associated files, all the usernames fields are configured as "ubuntu".

References

[1] Innovative Computing Laboratory University of Tennessee Knoxville. *HPL Algorithm*. 2018. URL: https://www.netlib.org/benchmark/hpl/algorithm.html (visited on 10/19/2023).

[2] Sébastien Rumley et al. "Evolving Requirements and Trends of HPC". In: *Springer Handbook of Optical Networks*. Ed. by Biswanath Mukherjee et al. Cham: Springer International Publishing, 2020, pp. 725–755. ISBN: 978-3-030-16250-4. DOI: 10.1007/978-3-030-16250-4_22. URL: https://doi.org/10.1007/978-3-030-16250-4_22.

[3] Erwan Sturzenegger. "Infrastructure de calcul pour l'institut iCoSys, la filière ISC, la HEIA voire au-delà". In: 2023.

[4] Kubeflow. *Kubeflow - The Machine Learning Toolkit for Kubernetes*. 2023. URL: https://www.kubeflow.org/ (visited on 10/09/2023).

[5] Inc. Advanced Clustering Technologies. *How do I tune my HPL.dat file?* 2024. URL: https://www.advancedclustering.com/act_kb/tune-hpl-dat-file/ (visited on 01/30/2024).

[6] Innovative Computing Laboratory University of Tennessee Knoxville. *HPL - A Portable Implementation of the High-Performance Linpack Benchmark for Distributed-Memory Computers*. 2018. URL: https://www.netlib.org/benchmark/hpl/index.html (visited on 10/19/2023).

[7] Martin Roch-Neirey. *PS5-MRN-Tech GitLab repository*. 2024. URL: https://gitlab.forge.hefr.ch/ps5-martin-roch-neirey/ps5-mrn-tech (visited on 01/31/2024).

[8] Oracle. *Oracle - Java download portal*. 2024. URL: https://www.oracle.com/java/technologies/downloads/ (visited on 01/31/2024).

[9] OpenMPI. *OpenMPI - Version 4.1*. 2024. URL: https://www.open-mpi.org//software/ompi/v4.1/ (visited on 01/31/2024).

[10] Sébastien Rumley. "Cours Infrastructures distribuées". In: 2023.

[11] OpenMathLib. *OpenBLAS - Version 0.3.24*. 2024. URL: https://github.com/OpenMathLib/OpenBLAS/releases (visited on 01/31/2024).

[12] Bernhard Enders (bgeneto). *build-install-compile-openblas GitHub repository*. 2024. URL: https://github.com/bgeneto/build-install-compile-openblas (visited on 01/31/2024).

[13] Wikipedia. *Wikipedia: Technical debt*. 2024. URL: https://en.wikipedia.org/wiki/Technical_debt (visited on 02/01/2024).

[14] LYSR. *LYSR - Artificial Intelligence Platform*. 2024. URL: https://lysr.ch/ (visited on 01/31/2023).

[15] HashiCorp. *Terraform documentation: local$_{f}ile$(Resource)*. 2024. URL: https://registry.terraform.io/providers/hashicorp/local/latest/docs/resources/file (visited on 02/01/2024).

Glossary

Infrastructure as Code Practice where the management and provisioning of computing infrastructure are automated and managed using code and software development techniques, rather than through manual processes. 9

Java Development Kit Set of tools for developing and compiling Java applications, including a complete JRE, compilers, and debuggers. 20

Java Runtime Environment Software package that allows the execution of Java applications, including the Java Virtual Machine (JVM) and necessary libraries. 20

Kubernetes Orchestrator used to deploy scalable workloads as containers. 1

OpenStack Open-source cloud computing platform that provides a set of software tools and components to build and manage public and private cloud infrastructure. 1

Type 1 Hypervisor Low-level and efficient software layer that runs directly on the physical hardware of a host machine to control and manage multiple virtual machines without the need for an underlying operating system.. 7

VMware IT company that provides different solutions like the vSphere / ESXi suite to virtualize computers. In this project, "VMware" refers to the virtualization cluster. 1