Report for Cloud Computing

VMs and Containers benchmarking

University Of Trieste
Data Science and Artificial Intelligence
Cloud Computing - Giuliano Taffoni, Ruggero Lot, Stefano Alberto Russo
Fall Semester 2024/2025

Jacopo Zacchigna
16.05.2025 - final

Contents

Figures

Tables

Listings

1 Introduction

1.1 Objective

This report aims at evaluating and comparing the performance of virtual machines (VMs) and containers in a controlled environment. Specifically, we use VirtualBox[1] [1] to deploy VMs and Docker[2] [2] to run containers, both operating Ubuntu Server[3] [3].

The experiment is structured into two phases: first, setting up a cluster of VMs and a parallel cluster of containers with equivalent resource constraints; second, executing a series of benchmarks to measure and analyze their performance.

1.2 Benchmark Overview

The following tools and methods were used to assess different performance dimensions:

- **CPU & Memory**:
  - High-Performance Linpack (HPL) and the HPC Challenge (HPCC) suite
  - **`sysbench`** and **`stress-ng`** for general system load testing
- **Disk I/O**:
  - **`IOZone`** and for local and shared disk performance
- **Network**:
  - **`iperf3`** for measuring throughput and latency between nodes

1.3 Host configuration

The host computer has the following specifications:

- **OS**: macOS Sequoia 15.4.1.
- **CPU**: Apple M4 Pro 12c/12t (8 Performance + 4 Efficiency). Base/Boost clock speed: 3.4/4.5 GHz (P-cores). Each core thread corresponds to one core.
- **RAM**: 24 GB Unified LPDDR5X. Fully shared between CPU/GPU Neural Engine.
- **Disk**: Apple NVMe SSD 512GB SSD.
- VBoxManage –version 7.1.8r168469
- Docker version 28.1.1, build 4eba377327

[1] https://www.virtualbox.org
[2] https://www.docker.com
[3] https://ubuntu.com/download/server

1.4 Project Scope

This report focuses on:

- Documenting the setup process for both virtualised and containerized clusters
- Presenting benchmark results for each test case
- Analyzing and comparing the outcomes across environments

> 
> The full set of benchmark data and scripts used in this analysis is available on GitHub, with custom README.md for each essentially each part of the process.

2 Virtual Machine Cluster Setup

What follows are the steps to set up and network the virtual machines on which we will later conduct performance testing.

> 
> For this, we setup VirtualBox. The tutorial[4] referenced throughout this section was instrumental in guiding the process.

This modified guide walks through setting up a VM cluster for cloud environment testing, which I used to perform the test.

2.1 Setting Up the Template VM

1. Create a new virtual machine in VirtualBox by clicking on *Machine* > *New*.
2. Enter the template as the name for the VM, select the storage directory on the host, and choose the Ubuntu 24.04.1 amd64 (since I'm on an apple silicon mac_book) server ISO image.
3. Skip the unattended installation process, as it may cause issues (or if you use it then you have at least to install OpenSSH).
4. Configure hardware, the settings used in this case are: **2GB RAM and 2 CPU** (adjust based on host capabilities).
5. Set the virtual hard disk size to 20GB.

[4] https://github.com/Foundations-of-HPC/Cloud-basic-2024/blob/main/Tutorials/VirtualMachine/README.md

2.1.1 Installing Ubuntu Server

After configuring the Virtual Machine, start it and follow the installation wizard.

1. Carefully select the language and keyboard layout
2. Choose the standard Ubuntu Server installation
3. Leave the network configuration at its default for now; we will adjust this later.
4. Leave the proxy address blank.
5. Select a suitable mirror address; the installer will test options.
6. For storage, use the “entire disk”.
7. Set up the user profile (e.g., username: user01, server name: template).
8. Enable the OpenSSH server for remote access.
9. Complete the installation, then shut down the VM and remove the ISO image.

2.1.2 Configuring the Template

Firstly, start the machine created earlier and log in with the credentials **`username: user01`** and **`password: test`** which were set up previously. Then test if the network connectivity is working correctly. Next update the software using the following command:

```
sudo apt update && sudo apt upgrade
```

It is important to install two additional packages, this can be done with this command:

```
sudo apt install net-tools gcc make
```

Finally, we have to shutdown the node:

```
sudo shutdown -h now
```

2.1.3 Configure VirtualBox Internal Network

Create a Host-Only network named “CloudBasicNet” with the following configuration:

```
Mask: 255.255.255.0
Lower Bound: 192.168.56.2
Upper Bound: 192.168.56.199
```

This will let us set the first ip address as the master ip address

2.2 Configuring the entire cluster (with automated script)

Due to many inconveniences and problems in the process of setting up the machines and for ease of use in the future, I have created a script to create the cluster. It can be found in the git repository[5].

The **`setup_node.sh`** script automates the VM setup process for both master and worker nodes. It handles VM cloning, network configuration, SSH setup, shared file-system, and service configuration.

2.2.1 Prerequisites

- VirtualBox installed
- sshpass to remove the need for interactive password prompts
- A template VM named **`template`** already created
- **Configuration directories:**

- **`master_config/`** — Contains configuration files for master node
- **`node_config/`** — Contains generic configuration for worker nodes

Those configuration can also be found inside the GitHub directory with all the other configurations and scripts.

2.3 Basic Usage

```
# Usage: ./setup_node.sh [node_name] [ssh_port] [password]
# More details are present when running ./setup_node.sh --help

# Set up master node (will use port 3022 for SSH by default)
./setup_node.sh master

# Set up worker nodes (using different SSH ports)
./setup_node.sh node-01 4022
./setup_node.sh node-02 5022
```

Listing 1 - Basic usage of setup_node.sh script

2.3.1 Advanced Usage

You can specify the node name, SSH port, and password:

```
./setup_node.sh [node_name] [ssh_port] [password]

# Example:
./setup_node.sh node-01 4022 custom_password
```

Listing 2 - Advanced usage of setup_node.sh script

[5] https://github.com/Jac-Zac/Cloud-Computing-Final-Exam/blob/main/VM_project/setup_node.sh

Note: The script has been written incrementally when setting up the cluster and then refined at the end to improve it drastically. It can now be easily used to completely recreate the environment from scratch, which is actually what I have done to perform the benchmarks.

2.3.2 Access Pattern

- **Master Node**: Direct SSH access from your local machine
- **Worker Nodes**: SSH access only through the master node

Connect to the master node first:

```
ssh -i ~/.ssh/id_ed25519 -p 3022 user01@127.0.0.1
```

Then from the master, connect to worker nodes:

```
ssh node-0n
```

2.4 Manual Setup of the VMs

An alternative set up is to build the cluster manually, more informations can be found on this provided tutorial. Which has been updated and now works without any changed to the DNS configuration.

Note that the following section showcases many commands that can be run through the terminal instead of interacting with the VirtualBox GUI. Those command are exactly what is being run under the hood in the automated script.

2.4.1 Cloning VMs

```
VBoxManage clonevm "template" --name "master" --register --mode all
VBoxManage clonevm "template" --name "node-01" --register --mode all
```

List all your virtual machines with: ***`VBoxManage list vms`***

2.4.2 Network Configuration

2.4.2.1 Configure Network Adapters

```
VBoxManage modifyvm "VM_NAME" --nic2 intnet
VBoxManage modifyvm "VM_NAME" --intnet2 "CloudBasicNet"
```

Replace **VM_NAME** accordingly.

2.4.2.2 Enable SSH Port Forwarding

```
VBoxManage modifyvm "master" --natpf1 "ssh,tcp,127.0.0.1,3022,,22"
VBoxManage modifyvm "node-01" --natpf1 "ssh,tcp,127.0.0.1,4022,,22"
```

This sets up port forwarding rules in VirtualBox NAT networking to forward SSH traffic from specific ports on the host (e.g., **3022**, **4022**) to port **22** (the default SSH port) on the guest VMs named **"master"** and **"node-01"**.

2.5 Master Node Configuration

2.5.1 SSH Key Setup

```
ssh-copy-id -i ~/.ssh/id_ed25519.pub -p 3022 user01@127.0.0.1
```

2.5.2 Copy Configuration Files

```
scp -P 3022 -r master_config user01@127.0.0.1:~
```

Finally Generate SSH key for to then connect to the other nodes easily:

```
ssh-keygen
```

2.5.3 Network Setup

This setup can be done leveraging the configuration files that are inside the master_config directory which we moved to the node

```
sudo cp ~/master_config/50-cloud-init.yaml /etc/netplan/50-cloud-init.yaml
sudo netplan apply

echo "master" | sudo tee /etc/hostname > /dev/null
sudo hostnamectl set-hostname master
sudo cp ~/master_config/hosts /etc/hosts
```

2.5.4 DNS and DHCP Setup

```
# Update package list and install dnsmasq
sudo apt update && sudo apt install dnsmasq -y

# Replace dnsmasq configuration with custom file
sudo cp ~/master_config/dnsmasq.conf /etc/dnsmasq.conf

# Ensure directory for additional dnsmasq configs exists
sudo mkdir -p /etc/dnsmasq.d

# Backup current resolver config and replace with custom one
sudo cp /etc/resolv.conf /etc/resolv.conf.backup
sudo unlink /etc/resolv.conf
sudo cp ~/master_config/resolv.conf /etc/resolv.conf

# Link systemd-resolved's resolv.conf for dnsmasq use
sudo ln -s /run/systemd/resolve/resolv.conf /etc/resolv.dnsmasq

# Restart services to apply changes
sudo systemctl restart dnsmasq
sudo systemctl restart systemd-resolved

# Enable dnsmasq to start on boot
sudo systemctl enable dnsmasq

# Reboot to finalize setup
sudo reboot
```

After bootstrap may happen the dnsmasq service start before the interfaces, just restart the service (the problem can be fixed with some configuration).

```
sudo systemctl restart dnsmasq systemd-resolved
```

Though I have added a way to fix the problem automatically inside **fix_dnsmasq_startup.sh** script

2.5.5 NFS Server Setup

Setting up a shared file system is essential in our project. We can manually do it performing the following actions

```
sudo apt install nfs-kernel-server -y

# Create shared directory for NFS exports
sudo mkdir -p /shared

# Set permissions to allow full access to all users
sudo chmod 777 /shared

# Add NFS export entry for the 192.168.56.0/24 subnet with specified options
echo '/shared/ 192.168.56.0/255.255.255.0(rw,sync,no_root_squash,no_subtree_check)' |
sudo tee -a /etc/exports

# Enable NFS server to start on boot and restart NFS server to apply export changes
sudo systemctl enable nfs-kernel-server
sudo systemctl restart nfs-kernel-server

# Create additional subdirectories inside the shared folder
sudo mkdir -p /shared/data /shared/home /shared/ssh-keys

# Set full access permissions on the new subdirectories
sudo chmod 777 /shared/data /shared/home /shared/ssh-keys
```

Note that we write **`192.168.56.0/255.255.255.0`** to specifies the allowed client network (all hosts in the 192.168.56.x subnet).

(rw,sync,no_root_squash,no_subtree_check) are options controlling how clients can access the share:

- rw: clients have read and write permissions.
- sync: writes to the shared directory are committed synchronously for data integrity.
- no_root_squash: remote root users retain root privileges on the share (not mapped to a less privileged user).
- no_subtree_check: disables subtree checking to improve reliability when exporting directories that may be moved or renamed.

This entry allows all machines in the specified subnet to mount and fully access the **`/shared/`** directory with the defined options.

2.6 Worker Node Configuration

We can now startup the machine copy the user configuration inside by coping the **`node_config`** directory and then ssh into the node to configure it

2.6.1 Initial Setup

```
scp -P 4022 user01@127.0.0.1:~/node_config ./node_config
ssh user01@node-01
```

2.6.2 Network Setup

Configure the network applying the configurations present on github.

```
sudo cp ~/node_config/50-cloud-init.yaml /etc/netplan/50-cloud-init.yaml
sudo netplan apply

echo "" | sudo tee /etc/hostname > /dev/null
sudo unlink /etc/resolv.conf
sudo cp ~/node_config/resolv.conf /etc/resolv.conf
sudo reboot
```

2.6.3 NFS Client Setup

Setting up a NFS client to automatically do the mounting of the shared file system at startup

```
sudo apt install nfs-common -y
sudo mkdir -p /shared/data /shared/home
```

2.6.3.1 AutoFS Setup

To mount it directly ot the shared directory we can configure it as followed. Which is also what was done in the automatic script.

```
sudo apt -y install autofs
echo '/-      /etc/auto.shared' | sudo tee -a /etc/auto.master > /dev/null
echo "/shared    master:/shared" | sudo tee -a /etc/auto.mount > /dev/null
sudo systemctl enable autofs
sudo systemctl restart autofs
```

2.6.4 Testing & Verification

To conclude the setup we can test the shared file system and check that we can correctly ping nodes by host name

```
touch /shared/data/test-from-node-01
ls -la /shared/data/

ping -c 3 master
```

2.7 Node Management

We can finalize the setup by creating a key on master and coping it to the shared file-system and then from the node directly add the key to the authorized keys

2.7.1 Node Status Script

Finally we can *create/use* the following script to check what nodes are running.

```bash
#!/bin/bash
n=${1:-9}
nodes=(master)
for i in $(seq 1 "$n"); do
  num=$(printf "%02d" "$i")
  nodes+=("node-$num")
done

for node in "${nodes[@]}"; do
  echo -n "$node: "
  ping -c 1 -W 1 "$node" >/dev/null && echo "UP" || echo "DOWN"
done
```

Listing 3 - Script check_node.sh: to get the machines that are up

Save scripts in **/shared/scripts/** for shared access

3 Container Setup

This section focuses on implementing a similar setup to what we have previously done using Virtual Machines but with containers. In this case the container engine used is Docker. The architecture similarly to before; consists of one **master** and two **worker nodes**. Communication between nodes is enabled by SSH, and MPI (Message Passing Interface) benchmarks are executed using a shared volume and predefined IP addresses.

The deployment consists of a master node and two worker nodes that execute master's direction on benchmark workloads together with the master. Moreover a shared volume which has the role of facilitating file sharing and finally predefined IP aliases to simplify communication between containers.

This setup, which can be deployed through the use of Docker Compose, allows an automated execution of MPI benchmarks.

3.1 Project structure

The project's directory layout is as follows:

```
.
├── Dockerfile              # Container image configuration
├── compose.yaml            # Multi-container orchestration
├── entrypoint.sh           # Initialization script
└── Performance_Testing/    # A directory with the tests to perform
```

This modular structure allows for straightforward benchmarking, logging and scalability while also supporting separation concerns.

3.2 Getting started

1. Make **`entrypoint.sh`** executable

```
chmod +x entrypoint.sh
```

2. Build and Start the Cluster through the use of Docker Compose

> ⚠ Each node has a limit of 2 CPU cores. However, core pinning for Docker containers using the **`--cpuset-cpus`** option is not very effective considering the virtualisation layer in between.

```
docker-compose -f compose.yaml up --build
```

The system builds automatically the images and starts three interconnected containers: the master node (**`master`**), which can ssh in other machines ad the two worker nodes (**`node-01`** and **`node-02`**).

The master generates an SSH key and shares it with workers via a Docker-managed volume **`hpc-shared`**.

3.3 SSH communication

> ℹ Following a similar structure to what has been done in VMS

Upon startup:

- The master node generates a root SSH keypair (if not already present).
- Worker nodes wait until the master's public key is available, then append it to their **`authorized_keys`**.
- Each container starts an SSH daemon, allowing for remote command execution.

3.4 Benchmarks execution

Once all containers are operational, benchmarks workloads can be executed from the master node:

```
docker exec -it master bash
cd /shared/Performance_Testing
./run-all.sh /config/mpi-hostfile
```

When running scripts with mpi if you haven't configured another user you might need to export the following flags in your shell:

```
OMPI_ALLOW_RUN_AS_ROOT=1
OMPI_ALLOW_RUN_AS_ROOT_CONFIRM=1
```

You can do so directly or put them inside your **`.bashrc`**/**`.zshrc`**.

We can get stats on our containers by running: **`docker stats`**

At this point it is important to ensure that the benchmark script **`run-all.sh`** exists and is executable.

The system has two shutdown modes:

1. Standard shutdown: stops the running containers but does not delete the volumes.

```
docker-compose down
```

2. Full cleanup: removes containers and cleans up all volumes (removes results and shared SSH files):

```
docker-compose down -v
```

3.4.1 Considerations

There are several considerations to address in the system design:

- Ensure ports (like SSH) are not blocked by local firewalls.
- Use the fixed IPs or container names to connect between nodes.
- Check logs with:

```
docker logs master
docker logs node-01
docker logs node-02
```

3.4.2 Health Check

- To verify service availability, the **`master`** container includes a health check to confirm that the SSH service is running:

```
healthcheck:
  test: ["CMD", "nc", "-z", "localhost", "22"]
  interval: 30s
  timeout: 10s
  retries: 3
```

And all the worker nodes specified this in their configuration:

```
depends_on:
  - master
```

This makes nodes services depends on the service named master. Thus instructing Docker Compose to start the master service before starting the dependent service. Therefore controlling the startup order of containers within the same **`compose`** file.

From the Github repo also the compose file and the Dockerfile are also accessible.

4 Benchmarks

To evaluate the various environments (virtual machines, dock containers and host systems) performance, a comprehensive benchmarking technique was used.

Benchmarks mainly focused on the following domains:

- CPU performance using **`hpcc`** and **`sysbench`**
- Memory performance using **`sysbench`**
- Disk I/O (both local and over shared file systems) with **`IOZone`**
- Network performance using **`iperf3`**
- High Performance Computing (HPC) workloads using MPI.

The aim was to compare performance and resource isolation among these key environments.

4.1 Environment configuration

The framework was deployed across the following environments: a host machine (macOS M4 in this case), three docker containers (a master and two workers) that simulate a HPC cluster, and multiple virtual machines with 2 vCPUs and 2 GB RAM.

Docker compose enabled the containerised cluster management. Thanks to a bridge network it established inter-container communication. Also, via a docker-managed volume mounted at **`/shared`**, shared filesystem access was enabled.

From this starting point the following test were performed both on the VirtualBox and the Docker cluster (some test were also performed on the host machine).

4.2 Installation

Dependencies can be installed using this script if not installed already:

```
./install-deps.sh
```

Folder structure to test different parts of the cluster

```
.
├── bin
│   ├── common.sh
│   ├── cpu-benchmark.sh
│   ├── disk-benchmark.sh
│   ├── mem-benchmark.sh
│   └── net-benchmark.sh
├── configs
│   ├── hpccinf.txt
│   └── mpi-hostfile
├── results/
├── errors.md
├── install-deps.sh
├── README.md
└── run-all.sh
```

4.3 CPU benchmarking

4.3.1 High-Performance Linpack (HPL)

The HPL benchmark is part of the **hpcc** package, its execution aims at assessing the floating-point computation performance across the different cluster nodes.

Firstly, we have to verify that all nodes are updated and that **hpcc** is installed. This can be done with the following command:

```
sudo apt update
sudo apt install hpcc
```

At this point, a hostfile that lists all node hostnames is created with the name **mpi-hostfile**.

The benchmark is run with default parameters to check the setup integrity:

```
mpirun.openmpi -mca btl_tcp_if_include enp0s9 -np 6 -hostfile mpi-hostfile hpcc
```

Finally, the benchmark was configured for a larger problem size with the example input file (**_hpccinf.txt**) customised as follows:

```
1024 2048 4096 8192   Ns         (problem size)
32 64 128 256         NBs        (block sizes)
2                     Ps         (process grid dimension P)
3                     Qs         (process grid dimension Q)
```

Thanks to this configuration, the work was tested on different problem sizes and spread across the cores of the different nodes.

4.3.2 Sysbench - CPU

In order to complement the HPL results, **sysbench** was used to benchmark the single-node CPU performance. Though this method lacks distributed execution support, the script was done in parallel on all nodes to showcase a more realistic cluster stress test.

We can verify the script is distributing correctly and see the version with the following commands

```
mpirun.openmpi -np 3 -hostfile mpi-hostfile sysbench --version
```

The next phase involves conducting comprehensive **CPU** tests. For this purpose, a custom script was developed to evaluate various components of the cluster. The script can be found on this github directory

It Contain tests such as the following:

```
log_info "-> Sysbench (max prime = 30k)"
sysbench cpu --cpu-max-prime=30000 --threads=2 run | tee -a "$RESULTS"
```

Run the script in parallel and log the info simply by running the following command:

```
# Running leveraging the auxiliary script
./run_all cpu config/mpi-hostfile
```

4.4 Memory performance

Similarly to what done for the CPU performance evaluation, memory benchmarks was also performed using **sysbench**:

```
log_info "-> Running sysbench memory test (500M)..."
sysbench memory --memory-block-size=1M --threads=2 --memory-total-size=500M run | tee -
a "$RESULTS"

log_info "-> Running stress-ng memory test (2 workers, 1 min)..."
stress-ng --vm 2 --vm-bytes 500M --timeout 60s --metrics-brief | tee -a "$RESULTS"
```

Again by running the custom script for the memory:

```
./run_all mem config/mpi-hostfile
```

4.5 Disk I/O performance

To evaluate the disk performance **IOZone** was used. With the disk-benchmark.sh script.

The test was performed as follows to test both the local and shared file-system:

```
#!/bin/bash

source "$(dirname "$0")/common.sh"

OUTPUT_FILE="${RESULTS:-disk_test_results.log}"
LOCAL_FILE="/tmp/iozone_local.tmp"
SHARED_MOUNT="/shared"
SHARED_FILE="$SHARED_MOUNT/iozone_shared.tmp"

# log_info "--- IOZone local filesystem test ---"
iozone -a -f "$LOCAL_FILE" 2>&1 | tee -a "$OUTPUT_FILE"
rm -f "$LOCAL_FILE"

if [[ -d "$SHARED_MOUNT" ]]; then
  log_info "--- IOZone shared filesystem test ---"
  iozone -a -f "$SHARED_FILE" 2>&1 | tee -a "$OUTPUT_FILE"
  rm -f "$SHARED_FILE"

else
  log_info "No shared filesystem found at $SHARED_MOUNT. Skipping shared tests."
fi
```

Listing 4 - Disk performance test

4.6 Network performance

To test inter-node bandwidth and latency, we use **iperf3** in both client and server modes.

Firstly, we install the tool and can start the server.

```
mpirun -np 3 --hostfile mpi-hostfile iperf3 --version
```

Afterwards, test were performed by having client on a node and a server on another one.

1. Master (server) ⟷ node (client)
2. Master (client) ⟷ node (server)
3. node-01 (server) ⟷ node-02 (client)

This test was performed using the net-benchmark.sh script

To set a node as a server you can run the following command:

```
# On server
iperf3 -s
```

4.6.1 HPC workload execution

Additional results for the **`hpcc`** test performed can be found on the Github repository with relative plots inside this directory

5 Benchmark results

5.1 HPL Benchmark Results

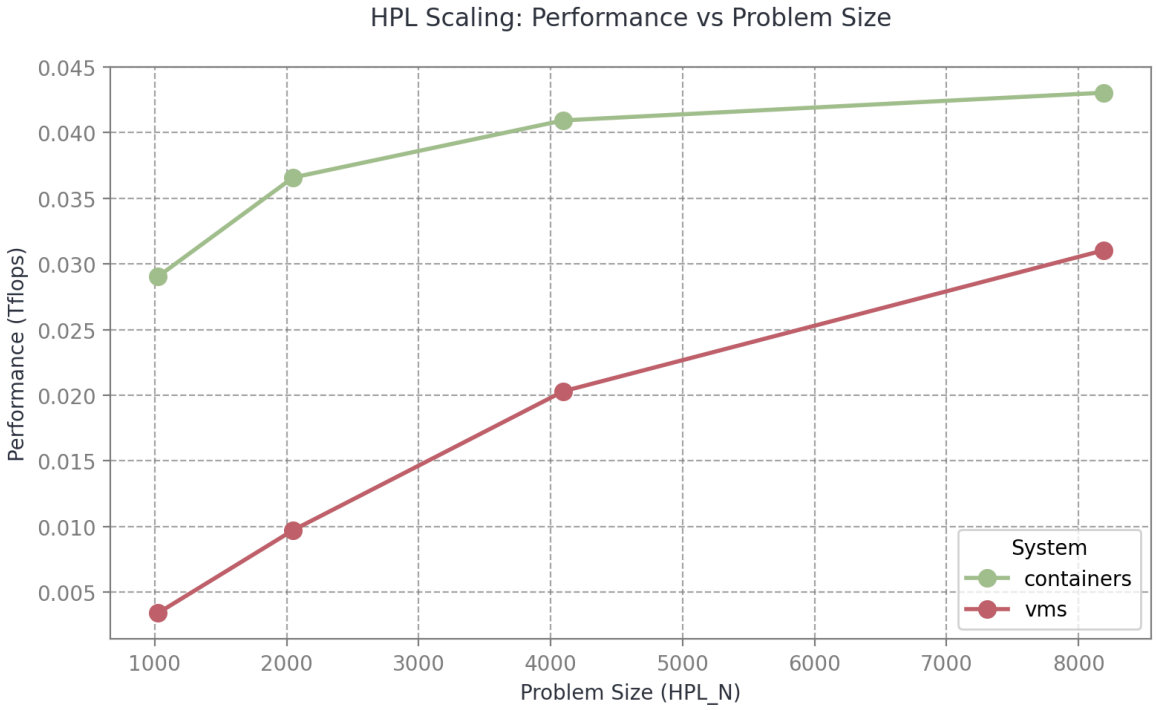

Figure 1 - HPL Performance Scaling Comparison

As shown in the scaling results across different problem sizes, the performance of virtual machines and containers is relatively similar. We observe an initial increase in Tflops starting from a problem size of **`1024`**, which is likely too small to fully stress the processor. After this, performance plateaus, but containers still demonstrate slightly better performance overall.

This outcome is expected, as both containers and VMs involve some degree of hardware virtualization-especially on macOS, where Docker containers run inside a lightweight VM. Therefore, the performance gap between the two is not very large.

5.2 CPU sysbench

In the following section we can see the results from sysbench.

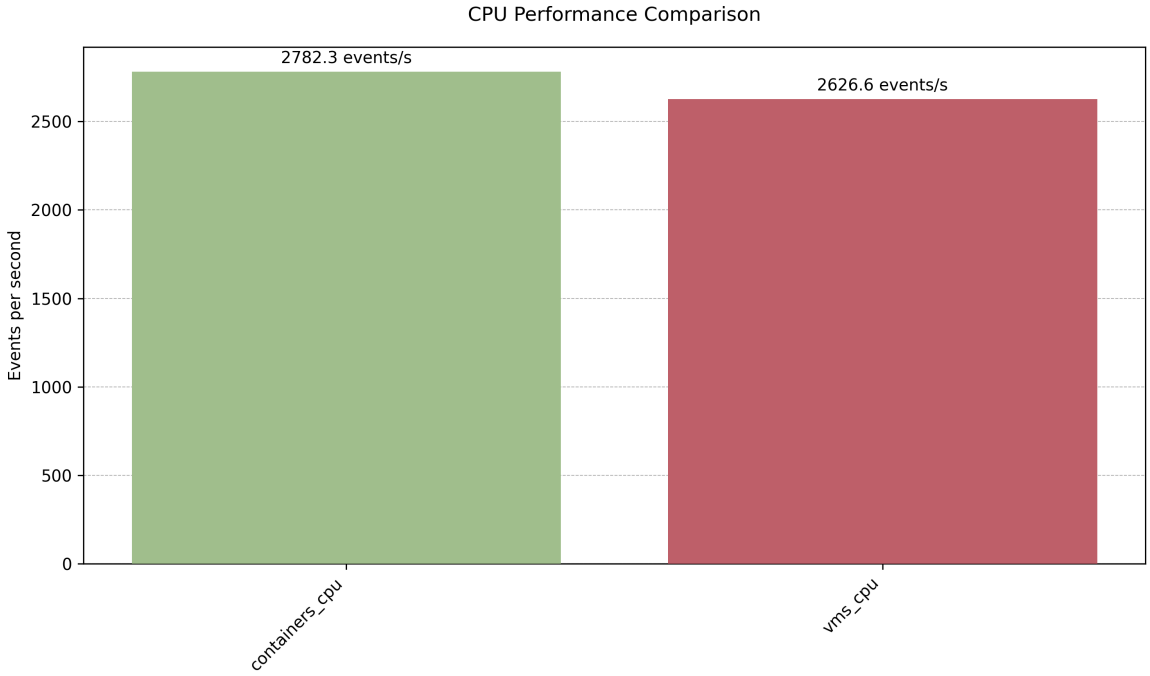

Figure 2 - CPU Performance Comparison Using Sysbench

,

	Event per second	Configuration
Host 2 CPU	34756540	standalone
VMs 2 CPU	2626.585	distributed
Containers 2 CPU	2782.27	distributed

Table 1 - Sysbench event per second comparison comparison

,

Indeed, the number of events per second is comparable between containers and virtual machines, with containers showing a slight advantage. The test was also conducted on the

host machine, but since it was not performed in a distributed setup, its results were not directly compared with the others. As expected though, the host machine's performance is significantly higher than both containers and virtual machines.

5.3 Memory test results

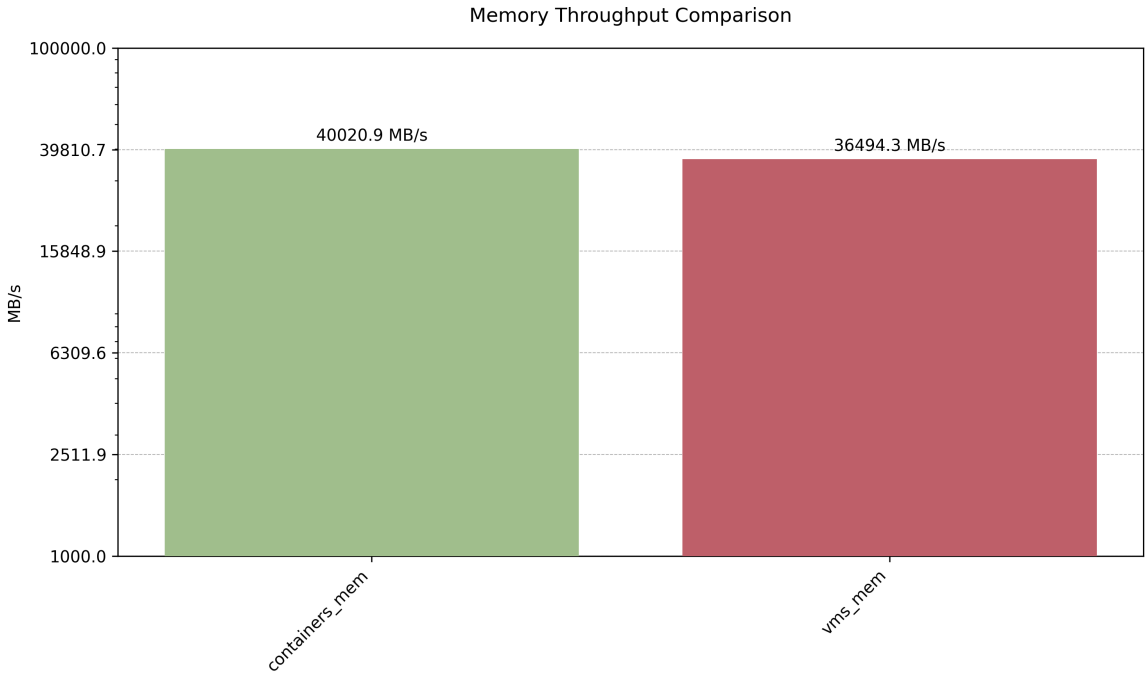

Figure 3 - Memory Performance Comparison Using Sysbench

,

	Memory MB/sec	Configuration
Host	58356.68	standalone
VMs	36494.305	distributed
Containers	40020.865	distributed

Table 2 - Sysbench Memory Throughput Comparison

,

The memory throughput results show that containers slightly outperform virtual machines in a distributed setup. Both virtualization approaches, however, deliver lower memory bandwidth compared to the host machine running standalone. This is expected due to the overhead introduced by virtualization layers. The host system achieves the highest memory throughput, indicating more direct and efficient access to physical memory resources.

5.4 Disk I/O test results

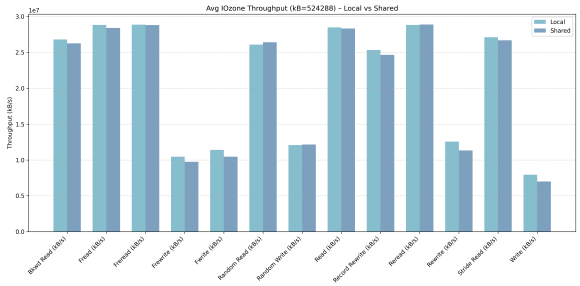

Figure 4 - Average Iozone Throughput

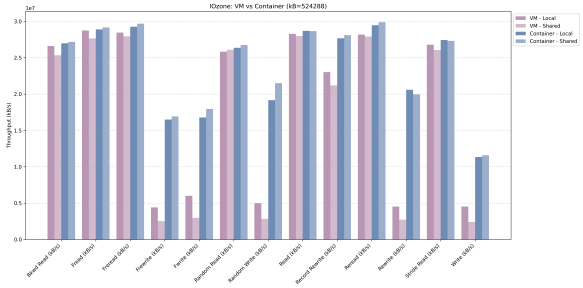

Figure 5 - Biggest Iozone Throughput

,

5.4.1 Disk Write Performance: Containers vs. VMs with Shared Storage

We compared disk write performance between containers and virtual machines (VMs) under two scenarios: local storage and shared storage. In our setup, VMs use NFS for sharing folders on Linux, while containers use a Docker named volume (**`hpc-shared`**), which is managed by Docker and typically mapped directly.

Key observations:

- *Write performance:* Containers consistently outperform VMs on shared storage writes. This is because Docker volumes avoid the overhead of network-based protocols like NFS, which require extra synchronization and introduce latency, especially for VMs.
- *Read performance:* Read speeds are similar between containers and VMs, regardless of the storage configuration. This is mainly because reads do not require the same level of synchronization or network communication as writes. Data can be fetched directly from the underlying storage, so the network stack (such as Ethernet or NFS) does not become a bottleneck for reads.

> ⚠ *macOS note:* On macOS, Docker containers themselves run inside a light-weight VM. This means container disk I/O performance is inherently similar to that of VMs, especially for read operations.

Docker containers with named volumes offer vastly superior shared storage performance compared to VMs using NFS. The main bottleneck for VMs is the NFS protocol overhead, while Docker volumes benefit from direct host filesystem access. This makes containers a more efficient choice for I/O-intensive workloads requiring shared storage.

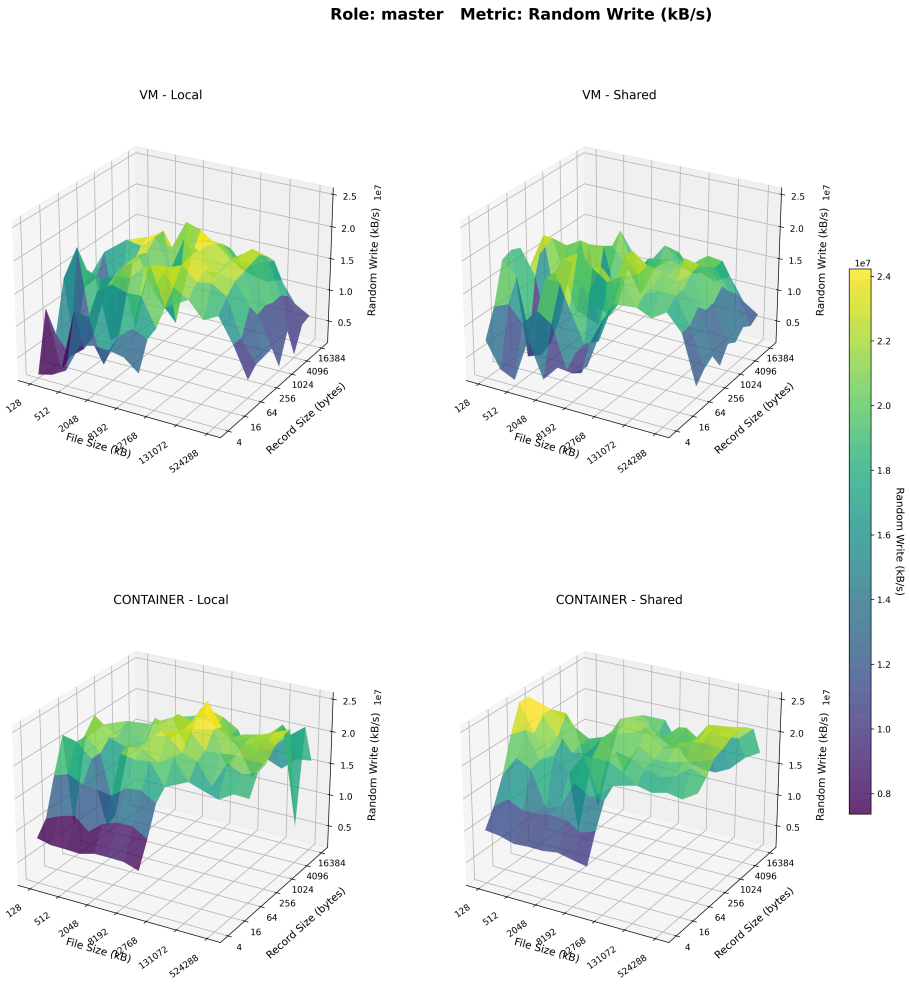

Figure 6 - Random Write Performance Vms vs Containers 3D

The figure above compares random write throughput for virtual machines (VMs) and containers, both with local and shared storage, across a range of file and record sizes.

We observe that *containers consistently deliver higher and more stable random write performance than VMs*, regardless of whether local or shared storage is used. This stability is particularly noticeable across varying file sizes, where containers exhibit less fluctuation in throughput.

For workloads sensitive to random write performance, containers provide a clear advantage over VMs, both in terms of absolute throughput and consistency across different I/O patterns.

5.5 Network test results

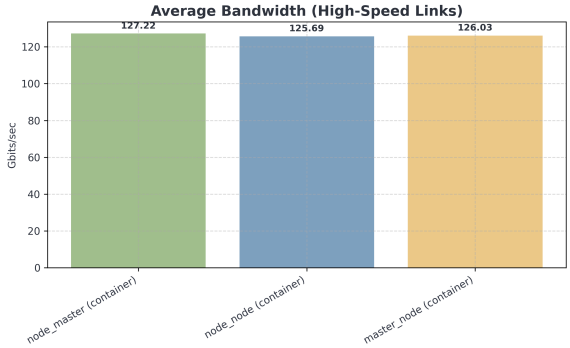

Figure 7 - Average BW High

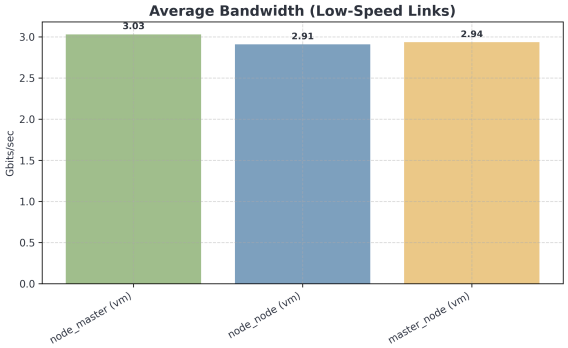

Figure 8 - Average BW Low

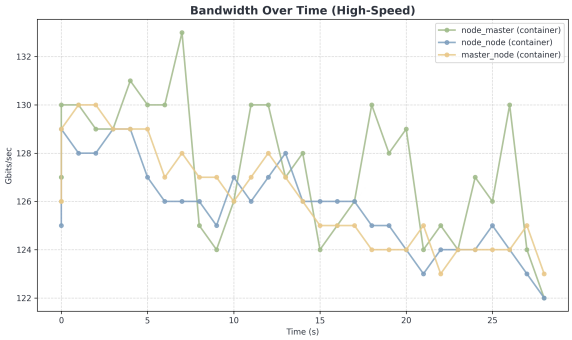

Figure 9 - BW TS High

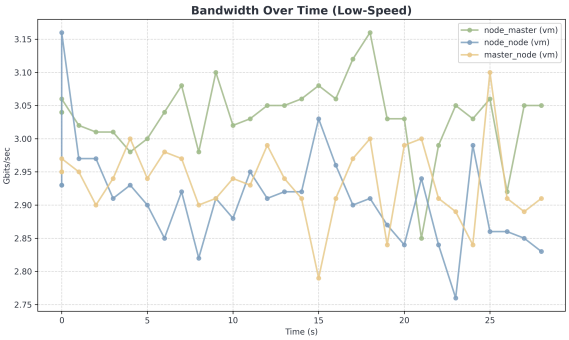

Figure 10 - BW TS Low

,

	Avg Bandwidth (Gbits/sec)	Avg Latency (ms)
node_master (container)	127.218750	0.11300
node_node (container)	125.687500	0.12388
master_node (container)	126.031250	0.11340
node_master (vm)	3.032188	0.26070
node_node (vm)	2.909687	0.32860
master_node (vm)	2.935313	0.30760

Table 3 - Network Bandwidth and Latency Comparison

,

The table compares average network bandwidth (in Gbits/sec) and latency (in ms) for communication between different host roles-node_master, node_node, and master_node-across two environments: containers and virtual machines (VMs).

5.5.1 Two Distinct Performance Groups

- *High Bandwidth (125-127 Gbits/sec) with Low Latency (0.11-0.12 ms):* Observed in all container-to-container communication pairs (node_master, node_node, master_node).
- *Low Bandwidth (2.9-3.0 Gbits/sec) with Higher Latency (0.26-0.33 ms):* Observed in all VM-to-VM communication pairs.

This clear division aligns with the fundamental differences between containers and VMs. Containers share the host OS kernel and communicate via virtual Ethernet interfaces entirely in memory, avoiding physical network interfaces. This results in very high throughput and low latency. Indeed this result is actually very similar to what was happening when making to processes communicate on the host with the same test.

5.5.2 Differences Among Node Pairs

All node pairs share the virtualization overhead inherent to VM networking. Differences among **`node_master`**, **`node_node`**, and **`master_node`** pairs might arise mainly from the master node's role in hostname/IP management, which can optimize routing and reduce latency/bandwidth penalties for communications involving the master.

Warning: Be very careful when performing tests like this. If you run **`iperf`** to test the master node from another node, it might resolve to **`localhost`** instead of the actual remote address. This can result in artificially high performance measurements that are not realistic.

The issue happens because the hostname *(e.g., "master")* resolves to a loopback IP (127.0.x.x) inside the VM. As a result, iperf3 sends traffic over the VM's loopback interface instead of the real virtual network interface. This causes the test to measure in-memory data copying rather than actual network throughput, leading to unrealistically high speeds. To fix this, use the VM's real IP address and ensure the hostname resolves correctly, or explicitly bind iperf3 to the proper interface. So instead of using master (the hostname) use the actaul IP address of the master node to test performances from a worker node.

6 Conclusions

To conclude, the benchmark analysis confirms what was expected. Containerised environments consistently outperform VMs across most areas. However, it is important to note that this project was done on a macOS system, which runs containers inside a virtualised Linux environment, introducing an additional layer of abstraction.

Thus, containers remain superior in performance and scalability compared to VMs. *This is probably because of the implementation of a very efficient lightweight (VM) running inside Docker Engine.* Though the overall efficiency is not as high as it would be on native Linux host. Indeed, CPU and memory benchmarks using sysbench show worse performance with virtual machines than containers, though both appear to be behind native host performance. In Disk I/O benchmarks the disparity between the two modalities is even bigger especially when dealing with shared file-systems. Indeed, containers, in particular those with docker-managed volumes, show much better write and consistency performance than VMs. The highest difference can be seen in the network benchmarks, where containers have achieved near-native bandwith and very low latency (below millisecond) because they are essentially just two process communicating one to the other. On the other hand, virtual machines show lower throughput and higher latency in communication between each other, because network traffic must pass through a virtual switch, which adds significant overhead.

Overall, containers demonstrated strong performance, though it would be interesting to test a true containerized macOS environment to evaluate it and compare it to running a virtualized Linux instance inside Docker Engine. Such a comparison could reveal the performance benefits of native containerization versus the virtualization overhead. VMs, due to their heavier virtualization stack, tend to exhibit performance penalties; however, they remain highly valuable as they enable emulation of different architectures, which can be essential in many scenarios.

Bibliography

[1] Oracle Corporation, “VirtualBox.” [Online]. Available: https://www.virtualbox.org/

[2] Docker, Inc., “Docker.” [Online]. Available: https://www.docker.com/

[3] Canonical Ltd., “Ubuntu Server 25.04 (Plucky Puffin).” [Online]. Available: https://ubuntu.com/download/server