Final Exam Exercise
Cloud Computing Performance Testing

Giovanni Lucarelli
Università di Trieste

May 15, 2025

https://github.com/giovanni-lucarelli/cloud-basic

Contents

1 Introduction

The objective of this project is to conduct a comparative performance evaluation of a cluster of nodes as Virtual Machines (VMs) and Containers, specifically focusing on VirtualBox and Docker. The evaluation will be based on a series of benchmarks that measure various performance metrics, including CPU, memory, disk I/O, and network throughput, namely: `hpcc`, `stressng`, `sysbench`, `IOZone`, `iperf`. Finally some of them will be discussed comparing the performance of the two supports and whenever possible also with the ones ont he sole host machine (no cluster).

2 Methodology

The entire project was performed on a laptop with the following specifications:

```
CPU Model: Intel(R) Core(TM) i7-8550U CPU @ 1.80GHz
Cores / Threads: 4 Cores / 8 Threads (4 cores x 2 threads/core)
RAM: 8 GB
Disk: SanDisk SD9SN8W2 (256 GB SSD)
Operating System: Ubuntu 24.04.2 LTS
Kernel Version: Linux 6.11.0-21-generic
```

2.1 Virtual Machine (VM) Setup

The cluster is built in Oracle VirtualBox (version 7.1.6r167084, avaiable on the official VirtualBox website) and it is made up by one master node `cluster01` and two worker nodes, namely `node01` and `node02`. The three of them are interconnected using an internal network and the workers inherited the Internet access through the master node, which act as DNS, DHCP and gateway of the cluster.

2.1.1 Template Machine Configuration

All VMs have identical specifications:

```
CPU: 2
RAM: 2048 MB
Disk: 20 GB
Operating System: Ubuntu 22.04.5 live server (amd64)
```

and they are built as clones of the same "template". After completing the installation of the OS in the template VMs, it has been configured with all the necessary software using the following commands:

```
sudo apt update && sudo apt upgrade
sudo apt install build-essential wget curl \
# network service
dnsmasq ssh iptables-persistent \
# benchmarking tools
iozone3 iperf3 netcat stress-ng sysbench openmpi-bin libopenmpi-dev hpcc
↪  \
nof-common nfs-kernel-server
# restarting the system
sudo shutdown -h now
```

The template VM has been cloned to create `cluster01` (master) and `node01` (worker); the second worker has been cloned from the first one after its conficuration.

Note: when cloning the template VM, it is important to select the option `Reinitialize the MAC address of all network cards` in order to avoid conflicts in the network configuration.

2.1.2 Network Adapters Configuration

The network configuration is done using the VirtualBox GUI. Two network adapters are created:

- **Adapter 1:** NAT, which allows the master to access the Internet through the host machine.
- **Adapter 2:** Internal Network, which allows the VMs to communicate with each other. To each VM is assigned a dynamic IP address.

The port forwarding is configured in the VirtualBox GUI, so that the SSH service can be accessed from the host machine. The following ports are used:

Name	Protocol	Host IP	Host Port	Guest Port
ssh	TCP	127.0.0.1	3333	22

The SSH service is configured to allow the host machine to connect to the VMs using the following command (after starting the master VM), on the host machine:

```
ssh-keygen
cd ~/.ssh
scp -P 3333 key_id.pub user01@127.0.0.1:~
```

In this way it is possible to connect to the master VM using the following command:

```
ssh -p 3333 user01@127.0.0.1
```

2.1.3 Master Node Configuration

Since the node has been copied from the template, it has the same username, password but also the same hostname. In order to avoid conflicts, the hostname of the master node has been changed using the following command:

```
sudo nano /etc/hostname
```

The hostname has been changed from `template` to `cluster01`. The same change has been done in the `/etc/hosts` file.

```
127.0.0.1 localhost
192.168.0.1 master
```

then the VM has been restarted in order to apply the changes. The master node is configured to act as DNS, DHCP and gateway of the cluster. The network interfaces available on the master node are:

Name	Type
enp0s3	NAT
enp0s8	Internal Network

The configuration is done using the `netplan` tool. The configuration file is located in `/etc/netplan/50-cloud-init.yaml` and it is as follows:

```
network:
  version: 2
  ethernets:
    enp0s3:
      dhcp4: true
    enp0s8:
      dhcp4: false
      addresses: [192.168.0.1/28]
      nameservers:
        addresses: [192.168.0.1]
```

Apply changes:

```
sudo netplan apply
```

Note: Disabling cloud-init ensures your netplan changes remain after reboot. `sudo touch /etc/cloud/cloud-init.disabled`

DNS Configuration (dnsmasq). The DHCP server is configured using the `dnsmasq` tool. The configuration file is located in `/etc/dnsmasq.conf` and it is as follows:

```
port=53
bogus-priv
strict-order
interface=enp0s8
listen-address=::1,127.0.0.1,192.168.0.1
bind-interfaces
log-queries
log-dhcp
dhcp-range=192.168.0.2,192.168.0.14,255.255.255.240,12h
dhcp-option=option:dns-server,192.168.0.1
dhcp-option=3
```

in order to control the way `dnsmasq` interacts with `resolv.conf`, the following lines have been uncommented in the `/etc/default/dnsmasq` file:

```
IGNORE_RESOLVCONF=yes
DNSMASQ_EXCEPT="lo"
```

the application of the changes is done using the following command:

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

Gateway Configuration. The gateway is configured by creating the file `etc/sysctl.d/99-sysctl.conf` with the following content:

```
net.ipv4.ip_forward=1
```

Apply:

```
sudo sysctl --system
```

To configure the IP tables, the following command is used:

```
sudo iptables -t nat -A POSTROUTING -o enp0s3 -j MASQUERADE
sudo netfilter-persistent save
```

Distributed File System (NFS). A shared directory is created in the master node using the following command:

```
sudo mkdir /shared
sudo chmod 777 /shared
sudo nano /etc/exports
```

The content of the `/etc/exports` file is as follows:

```
/shared/
↪  192.168.0.0/255.255.255.240(rw,sync,no_root_squash,no_subtree_check)
```

and then the NFS service is enabled and restarted using the following command:

```
sudo systemctl enable nfs-kernel-server
sudo systemctl restart nfs-kernel-server
```

2.1.4 Worker Nodes Configuration

Analogously to the master node, the hostname of the worker nodes has been changed using the following command:

```
sudo nano /etc/hostname
```

The hostname has been changed from `template` to `node01`.

Network Configuration The network configuration file, located in `/etc/netplan/50-cloud-init.yaml`, it has been modified as follows:

```
network:
  ethernets:
    enp0s8:
      dhcp4: true
      dhcp-identifier: mac
      nameservers:
        addresses: [192.168.0.1]
      routes:
        - to: 0.0.0.0/0
          via: 192.168.0.1
```

and applied using `sudo netplan apply`. The DNS server has been configured using:

```
sudo ln -fs /run/systemd/resolve/resolv.conf /etc/resolv.conf
```

SSH setup. In order to access the worker nodes through the master (not from the host, because of the internal network), the SSH service has been configured using the following command:

```
// from master node
ssh-keygen -t rsa -b 4096
ssh-copy-id node01
```

after doing this it is possible to access the worker nodes using `ssh node01`

Distributed File System Configuration. To create a mount point for the shared directory, the following command is used:

```
sudo mkdir /shared
```

then to Mount the shared directory from the master node to this folder:

```
sudo mount 192.168.0.1:/shared /shared
```

and to make the mount persistent across reboots the package AutoFS is installed

```
sudo apt -y install autofs
```

then in the `auto.master` configuration file the following line has been added in order to include the mount point

```
/- /etc/auto.mount
```

In order to define the NFS mount the AutoFS configuration file (`auto.mount`) has been created and the following line has been added:

```
/shared -fstype=nfs,rw  192.168.0.1:/shared
```

and to apply changes

```
sudo systemctl restart autofs
```

Clone the Worker Node. The second worker node has been cloned from the first one after its configuration. The hostname of the second worker node has been changed in the usual way in `node02`.

2.2 Container Setup

The Docker service (version `28.1.1`, build `4eba377`) has been installed according to the documentation available at Docker official website.

2.2.1 Template Machine Configuration (Dockerfile)

The Dockerfile is used to create the image of the container. The content of the `Dockerfile` is as follows:

```
FROM ubuntu:latest

ENV DEBIAN_FRONTEND=noninteractive

RUN apt-get update && apt-get install -y \
    build-essential wget curl \
    openssh-server rsync iputils-ping \
    dnsmasq ssh iptables-persistent \
    iozone3 iperf3 netcat-openbsd \
    stress-ng sysbench openmpi-bin libopenmpi-dev hpcc \
    nfs-common nfs-kernel-server sudo \
    && rm -rf /var/lib/apt/lists/*

# Add the user
RUN useradd -m -s /bin/bash user01 && echo "user01:0000" | chpasswd \
    && echo "user01 ALL=(ALL) NOPASSWD:ALL" >> /etc/sudoers

# Create SSH directory and generate keys for user01
RUN mkdir -p /home/user01/.ssh && \
    ssh-keygen -t rsa -N '' -f /home/user01/.ssh/id_rsa && \
    cat /home/user01/.ssh/id_rsa.pub > /home/user01/.ssh/authorized_keys
    ↪  && \
    chmod 700 /home/user01/.ssh && \
    chmod 600 /home/user01/.ssh/id_rsa /home/user01/.ssh/authorized_keys
    ↪  && \
    chown -R user01:user01 /home/user01/.ssh

# Copy the public key to a shared location (optional)
RUN cp /home/user01/.ssh/id_rsa.pub /tmp/user01_rsa.pub

# Create the directory for privilege separation
RUN mkdir -p /run/sshd

# Generating the host keys is important for SSH to work properly
# The -A flag generates all the default keys
RUN ssh-keygen -A

# Expose the SSH port
EXPOSE 22
```

```
# Switch to user01
USER user01
WORKDIR /home/user01

# Start the SSH server
# The -D flag runs the server in the foreground
# The -e flag enables logging to stderr
CMD ["sudo", "/usr/sbin/sshd", "-D", "-e"]
```

2.2.2 Cluster Configuration (Docker Compose)

The cluster is built using Docker Compose. The `docker-compose.yml` file is as follows:

```
services:
  # Define the services (containers) that make up the cluster.

  cluster01:
    # Build the Docker image for this service using the Dockerfile in the
    ↪  current directory (.).
    build: .
    container_name: cluster01

    # Set the hostname inside the container. Used for communication using
    ↪  hostnames
    hostname: cluster01

    networks:
      # Connect this container to the 'internal-net' defined below
      internal-net:

    deploy:
      # Define resources (CPU = 2, memory limits = 2GB)
      resources:
        limits:
          cpus: "2"
          memory: 2G

    # Map ports from the host machine to the container.
    # allows to SSH into the container from the host using 'ssh -p 2220
    ↪  user@host'.
    ports:
      - "2220:22"

    # Mount shared volume into the container for sharing data.
    volumes:
      - shared-data:/shared
```

```
  # Define the first worker/node service, configured similarly to
  ↪  cluster01.
  node01:
    build: .
    container_name: node01
    hostname: node01
    # Specify that this container should be started after the 'cluster01'
    ↪  service
    # to create a similar structure w.r.t. VMs
    depends_on:
      - cluster01
    networks:
      internal-net:
    deploy:
      resources:
        limits:
          cpus: "2"
          memory: 2G
    ports:
      - "2221:22"
    volumes:
      - shared-data:/shared

  # Define the second worker/node service, configured similarly.
  node02:
    build: .
    container_name: node02
    hostname: node02
    depends_on:
      - cluster01
    networks:
      internal-net:
    deploy:
      resources:
        limits:
          cpus: "2"
          memory: 2G
    ports:
      # Map host port 2222 to port 22 inside this container.
      - "2222:22"
    volumes:
      - shared-data:/shared

# Define the networks used by the services.
networks:
  internal-net:
    # 'bridge' is the default and creates a private internal network on
    ↪  the host machine,
    # that allows containers on this network to communicate with each
    ↪  other using their hostnames
```

```
    driver: bridge

# Define the (shared) volume
volumes:
  shared-data:
    # Specify the volume driver. 'local' is the default and stores the
    ↪  volume
    # data in a directory on the host machine managed by Docker.
    driver: local
```

Note: there is a slight difference between the cluster configuration in VirtualBox an the one in Docker. In VirtualBox the master node has two network adapters, one for the Internet access and one for the internal network, while in Docker the cluster is built using a single network adapter. The master node has a public IP address (the host machine) and the worker nodes have private IP addresses (the internal network). The shared directory is mounted in each container using a volume. This difference however should not affect the performance of the chosen tests.

2.2.3 Starting the Cluster

The cluster is started using the following command:

```
docker compose up -d --build
```

This command will build the images and start the containers in detached mode. The containers can be accessed using the following command:

```
docker exec -it cluster01 bash
# or
ssh -p 2220 user01@localhost
```

In order to give the `user01` the permission to access the shared directory, the following command is used:

```
sudo chown -R user01:user01 /shared
```

In order to access the each node from each other, the SSH service has been configured manually using , from `cluster01` for example:

```
ssh node01
exit
ssh node02
exit
```

this should be done at each restart of the containers, and analogously for the worker nodes in order to have all the nodes fully connected.

Two useful commands to manage the docker containers are: `docker ps` to see all the running containers and `docker stop $(docker ps -q)` to shut down all of them.

2.3 Testing Procedures

The following benchmarking tools have been used to perform the tests:

- **HPC Challenge**: A comprehensive suite of tests designed to measure the performance of high-performance computing systems. It evaluates both memory and computational performance using a variety of tests like HPL (High Performance Linpack), DGEMM, FFT, and STREAM.[1]
- **Stress-ng**: A tool for stress-testing CPUs, memory, I/O, and other system components. It's useful for validating system stability under heavy load by running various stressors in parallel.
- **Sysbench**: A modular benchmarking tool for evaluating system parameters such as CPU, memory and disk I/O.
- **Iozone**: A filesystem benchmark tool used to measure I/O performance across various operations such as read, write, re-read, and random access.
- **Iperf**: A network testing tool used to measure bandwidth and throughput between two endpoints over TCP or UDP.

The measurements were performed multiple times in each environment – VMs, containers and on the host whenever possible – in order to have also a measure of the variability of each benchmark. Since every environment runs the same operating system, the same script have been used across all the environment; moreover, during each measurement process has been ensured that no other heavy processes were running on the host machine during the tests.

Since we're testing the cluster environment and most of the tests need passwordless SSH protocol (from master to workers) is a good practice to run the following command before running the tests (both no VMs and containers) to be sure that everything works fine:

```
mpirun -np 4 -hostfile hosts hostname
```

and the expected output should be:

[1] https://hpcchallenge.org/hpcc/faq/index_print.html

```
node01
node01
node02
node02
```

2.3.1 HPCC

All the tests have been runned in the `/shared` folder, and the first step to run such benchmark is to locate the `hpcc` executable file and to copy it (and all the relative configurations file) in the former directory using:

```
which hpcc
cp -r /path/to/hpcc /shared/hpcc
```

The configuration file for the HPCC, namely `hpccinf.txt` has been obtained consulting the website reccomended during the lessons.

```
HPLinpack benchmark input file
Innovative Computing Laboratory, University of Tennessee
HPL.out      output file name (if any)
6            device out (6=stdout,7=stderr,file)
1            # of problems sizes (N)
20352         Ns
1            # of NBs
192           NBs
0            PMAP process mapping (0=Row-,1=Column-major)
1            # of process grids (P x Q)
2            Ps
2            Qs
16.0         threshold
1            # of panel fact
2            PFACTs (0=left, 1=Crout, 2=Right)
1            # of recursive stopping criterium
4            NBMINs (>= 1)
1            # of panels in recursion
2            NDIVs
1            # of recursive panel fact.
1            RFACTs (0=left, 1=Crout, 2=Right)
1            # of broadcast
1            BCASTs (0=1rg,1=1rM,2=2rg,3=2rM,4=Lng,5=LnM)
1            # of lookahead depth
1            DEPTHs (>=0)
2            SWAP (0=bin-exch,1=long,2=mix)
64           swapping threshold
0            L1 in (0=transposed,1=no-transposed) form
0            U  in (0=transposed,1=no-transposed) form
```

```
1            Equilibration (0=no,1=yes)
8            memory alignment in double (> 0)
##### This line (no. 32) is ignored (it serves as a separator). ######
0                               Number of additional problem sizes for
↪  PTRANS
1200 10000 30000                values of N
0                               number of additional blocking sizes for
↪  PTRANS
40 9 8 13 13 20 16 32 64        values of NB
```

then the file `hosts` has been created:

```
node01 slots=2
node02 slots=2
```

defines the nodes in which the hpcc benchmark will be executed in a distributed environment, specifying also the number of slots for each node, *i.e.*, the number of processors (logical or physical) available for the parallel execution of the MPI processes. The test has been repeated three times using the command:

```
mpirun -np 4 -hostfile hosts hpcc
```

2.3.2 stress-ng

Bash script for the `stress-ng` test:

```
#!/bin/bash

# Number of repetitions
REPEAT=5

# Hostfile location
HOSTFILE="hosts"

# Timeout for each test
TIMEOUT="60s"

# Loop over 5 repetitions
for i in $(seq 1 $REPEAT); do
    echo "Running CPU test iteration $i..."
    mpirun -np 4 -hostfile $HOSTFILE stress-ng --cpu 1 --timeout $TIMEOUT
    ↪  --metrics-brief \
        2>&1 | tee "stress_ng_cpu_run_${i}.txt"

    echo "Running VM test iteration $i..."
```

```
    mpirun -np 4 -hostfile $HOSTFILE stress-ng --vm 1 --vm-bytes 512M
    ↪  --timeout $TIMEOUT --metrics-brief \
        2>&1 | tee "stress_ng_vm_run_${i}.txt"

    echo "Finished iteration $i"
    echo "-------------------------"
done

echo "All tests completed."
```

the bash script has been saved as `run_stress_tests.sh` and then executed using

```
chmod +x run_stress_tests.sh
./run_stress_tests.sh
```

2.3.3 sysbench

Bash script for the `sysbench` test:

```
#!/bin/bash

# Number of iterations
ITERATIONS=5

# Run CPU test 5 times
echo "Running CPU test $ITERATIONS times..."
for i in $(seq 1 $ITERATIONS); do
    echo "CPU Test Run #$i"
    mpirun -np 4 -hostfile hosts sysbench --test=cpu
    ↪  --cpu-max-prime=20000 run | tee "sysbench_cpu_result_$i.txt"
done

# Run Memory test 5 times
echo "Running Memory test $ITERATIONS times..."
for i in $(seq 1 $ITERATIONS); do
    echo "Memory Test Run #$i"
    mpirun -np 4 -hostfile hosts sysbench --test=memory
    ↪  --memory-total-size=10G run | tee "sysbench_mem_result_$i.txt"
done

echo "All tests completed."
```

it has been saved as `run_sysbench_tests.sh` and then executed analogously to the previous test.

2.3.4 iozone

Local filesystem. In order to test the I/O of the local filesystem, from the `/shared` directory the following command has been executed

```
iozone -a -R -O | tee iozone_results.txt
```

the flag `-a` runs automatic mode, testing many combinations of file sizes and record sizes; `-R` Outputs results in Excel spreadsheet format; `-O` Adds OPS (operations per second) to the output.

Shared filesystem. To test the shared filesystem, an empty file has been created in the `/shared/iozone` directory. This is the target file Iozone will use for its I/O benchmarking.

```
touch /shared/iozone/testfile
```

Then the `machines.txt` has been created, containing all the nodes that will participate in the distributed Iozone test, and tells Iozone where it and the shared path are located:

```
node01 /shared/iozone /usr/bin/iozone
node02 /shared/iozone /usr/bin/iozone
```

The test has been runned using:

```
export ssh=rsh
iozone -+m /shared/iozone/machines.txt -f /shared/iozone/testfile -a -R
↪  -O | tee iozone_shared_results.txt
```

2.3.5 iperf

The network test has been performed several times, in order to test all the possible pair of connection between nodes, *i.e.*, (`cluster01`, `node01`), (`cluster01`, `node02`), (`node01`, `node02`). On the server node:

```
sudo killall iperf3 # clean up before running a test
iperf3 -s | tee iperf3_results.txt
```

the flag `-s` puts the program into **server mode**. On the client node:

```
iperf3 -c cluster01 # TCP test (upload)
iperf3 -c cluster01 -R # TCP Reverse test (download)
iperf3 -c cluster01 -u -b 3G # UDP test (upload)
```

the flag `-c` puts the program into **client mode**. With this commands are tested two core network communication protocols, namely TCP (upload and download) and UDP (upload). The flag `-b` is used to set the target bandwidth, the default value is 1 Mb/s for UDP and unlimited for TCP. The value of 3 Gb/s is used to test the maximum bandwidth of the network, after some trials to find the value in order to saturate the channel.

3 Results and Discussion

3.1 HPCC

As already mentioned before, the `hpcc` test consists in a large number of benchmarks; the full resoults can be found in the folder results in the github repository of the project. For clarity sake here are reported and discussed only the principal ones, divided according to the hardware stressed.

3.1.1 Compute Performance

This section evaluates the computational performance of the system across different environments. Specifically, the benchmark reported are: HPL (High-Performance Linpack), DGEMM (Double-precision General Matrix Multiply), and FFT (Fast Fourier Transform). All of them aim to assess floating-point computation in linear algebra and numerical problems. As presented in Table 2 and in Figure 1, the results consistently show that the container environment performs comparably to or exceeds the host environment across most metrics, while the virtual machine (VM) configuration trails behind.

A notable observation is the significant performance disparity exhibited by the VMs in the MPI FFT benchmark. Parallel FFT algorithms are inherently communication-intensive, frequently requiring extensive inter-rank communication, typically involving all-to-all exchanges for data transposition and redistribution. Consequently, beyond raw floating-point capability, this workload heavily stresses: network latency and bandwidth and MPI transport efficiency. A critical distinction among the three environments lies in their respective MPI communication mechanisms. On the host system, MPI communication between processes on the same node primarily utilizes highly efficient shared memory. In contrast, the container environment employs the TCP network protocol for inter-process communication, albeit potentially with lower overhead compared to VMs due to reduced virtualization layers. The VM environment, however, relies on TCP communication routed through virtual Network Interface Controllers (vNICs). This virtualization layer for networking introduces substantial overhead, resulting in significantly poorer latency and bandwidth characteristics for inter-process communication within and between VMs. These detrimental effects of virtualized networking on communication performance are clearly corroborated by the results of the ping-pong and iperf network benchmarks, presented in Table 4 and Table 7, respectively, which demonstrate substantially higher latency and lower bandwidth for the VM environment.

Benchmark	VM	Container	Host
HPL (Gflops)	5.128 ± 0.013	5.552 ± 0.034	5.516 ± 0.027
DGEMM (Gflops)			
StarDGEMM	1.290 ± 0.045	1.303 ± 0.122	1.400 ± 0.005
SingleDGEMM	1.907 ± 0.077	2.081 ± 0.052	2.054 ± 0.024
FFT (Gflops)			
StarFFT	1.453 ± 0.033	1.832 ± 0.025	1.816 ± 0.008
SingleFFT	2.638 ± 0.047	2.767 ± 0.054	2.739 ± 0.141
MPIFFT	1.231 ± 0.064	3.673 ± 0.124	4.042 ± 0.014

Table 2: Benchmark performance (in Gflops) across the three execution environments. Values are reported as mean ± standard deviation, calculated over three measurements ($n = 3$).

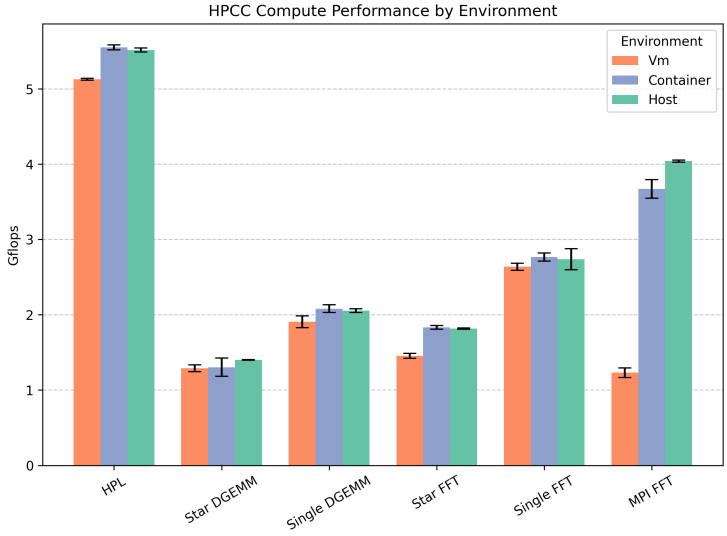


Figure 1: Compute Performance Benchmarks: VM (orange) vs Container (blue) vs Host (green). All the benchmarks are reported in Gflops.

3.1.2 Memory Benchmarks

The main memory benchmark results are presented in Table 3 and Figure 2. The benchmarks include SingleSTREAM, StarSTREAM, RandomAccess, and PTRANS. Each benchmark is designed to evaluate different aspects of memory performance, including bandwidth and latency.

STREAM Memory Bandwidth The STREAM benchmark measures sustainable memory bandwidth (typically reported in GB/s) through simple vector operations: *Copy*, *Scale*, *Add*, and *Triad*. The benchmark was executed in two configurations: SingleSTREAM, assessing memory performance on a single compute node, and StarSTREAM, which aggregates memory bandwidth across multiple nodes in a distributed setting.

In SingleSTREAM evaluations, the host and container environments demonstrated comparable memory bandwidth, while the virtual machine (VM) configuration consistently exhibited lower performance. Despite the VM's relative underperformance, all environments achieved a substantial percentage of the theoretical maximum bandwidth, indicating generally efficient operation of the local memory subsystems, with the exception of the VM, which is subject to certain virtualization-induced memory overheads.

To determine the nominal memory bandwidth of the system, hardware information was queried using the command `sudo dmidecode --type memory`. The relevant output is presented below:

```
Configured Memory Speed: 1867 MT/s
Bus width per channel: 64 bits = 8 bytes
Number of channels: 2
```

Based on this information, the theoretical peak memory bandwidth can be calculated as follows:

$$\text{Bandwidth} = 1.867\frac{\text{GT}}{\text{s}} \times 8\frac{\text{B}}{\text{T}} \times 2 = 29.872\frac{\text{GB}}{\text{s}}$$

Among the STREAM operations, the *Copy* operation – being highly optimized by compilers and frequently implemented using assembly code – provides the most direct measure of effective memory throughput. The observed throughput for the *Copy* operation was approximately 75% of the nominal bandwidth for the VM, 81% for the container, and 78% for the host. These results indicate generally satisfactory performance across all environments.

In the **StarSTREAM** benchmark the container environment again surpassed the VM setup. As data exchange between nodes necessitates significant network communication and synchronization, the effective memory bandwidth becomes constrained also by network bandwidth and latency rather than only local memory speed. Consequently, performance declines significantly compared to SingleSTREAM, with the observed bandwidth reaching only approximately

20% of the nominal maximum. This performance degradation is particularly pronounced in the VM cluster, probably due to the overhead associated with the virtualized of the network interface.

RandomAccess (RA) Benchmark The RandomAccess benchmark quantifies the rate of random memory updates, expressed in mega updates per second (MUP/s). Unlike STREAM, which focuses on sequential memory access patterns, RandomAccess evaluates the system's efficiency in handling numerous small, non-contiguous memory accesses.

In the MPI implementation of this benchmark, the host environment exhibited disproportionately high performance. This can be interpreted analogously to the case of MPI FFT and so can be attributed to the utilization of shared memory for inter-process communication on the host, which is substantially more efficient than the network-based communication employed in the VM and container environments.

PTRANS Benchmark The PTRANS benchmark assesses global memory bandwidth in parallel computing systems by measuring the rate at which large matrices can be transposed across processor nodes. It effectively gauges the system's capability to efficiently move large volumes of data between memory spaces on different nodes. Results from this benchmark further reinforce the significant impact of network performance on distributed memory operations. The container cluster demonstrated strong performance, whereas the VM cluster exhibited severe limitations, likely attributable to the same network bottlenecks identified in the StarSTREAM and MPI benchmarks. As shown later we will see that the main bottleneck of the virtual machines environment lies exactly in the network.

3.1.3 Latency and Bandwidth

The ping-pong benchmark is a fundamental test used to measure the performance of a communication channel or link between two endpoints. In a cluster environment, a ping-pong benchmark is typically run between two different nodes. On a single host machine, a ping-pong benchmark is typically run between two processes or threads running on that same machine. For this reason here the Host results are not a proper baseline to compare the VMs and container results.

From the table 4 we can see that VM introduces massive latency penalties with respect to the Containers. Analogously, for the bandwidth the containers show the best performances.

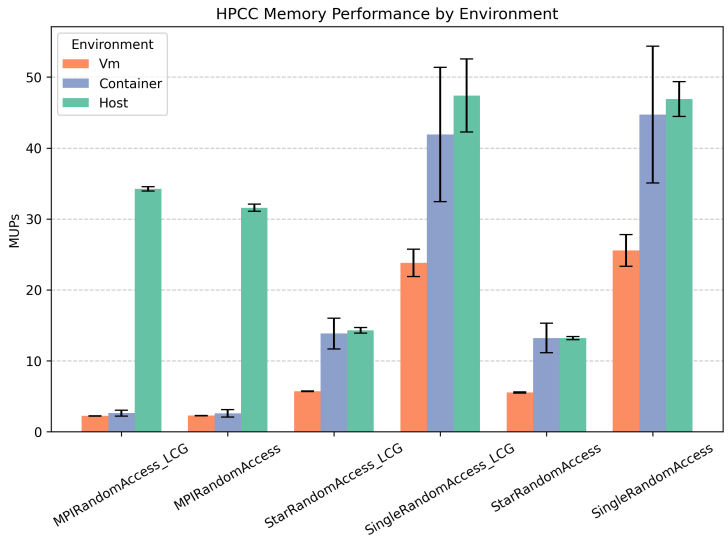


Figure 2: Memory Random Access Benchmarks: VM (orange) vs Container (blue) vs Host (green). All the benchmarks are reported in mega updates per second (MUP/s).

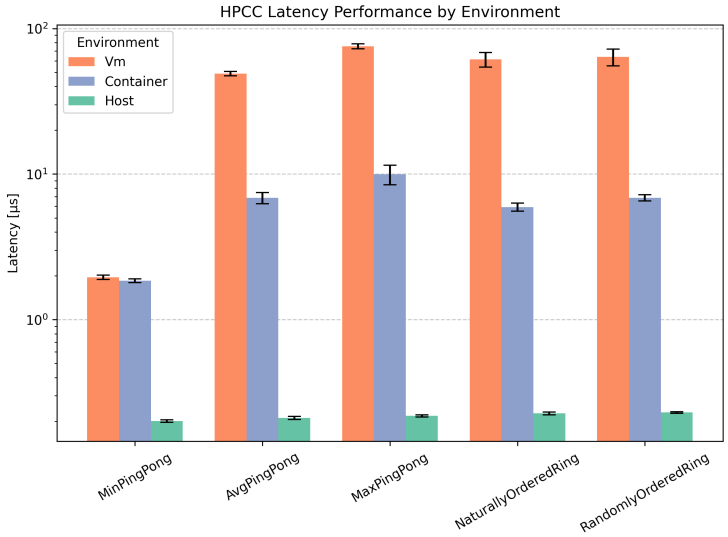


Figure 3: Latency and Bandwidth Benchmarks: VM (orange) vs Container (blue) vs Host (green). All the latencies are reported in microseconds (µs) and y-axis is in log scale.

Benchmark	VM	Container	Host
SingleSTREAM (GB/s)			
Copy	22.30 ± 0.32	24.11 ± 0.20	23.44 ± 0.06
Scale	13.26 ± 0.19	14.23 ± 0.06	14.06 ± 0.12
Add	14.40 ± 0.24	15.38 ± 0.16	15.06 ± 0.14
Triad	14.44 ± 0.28	15.48 ± 0.13	15.22 ± 0.05
StarSTREAM (GB/s)			
Copy	5.03 ± 0.03	5.41 ± 0.03	5.39 ± 0.02
Scale	3.34 ± 0.03	3.55 ± 0.01	3.56 ± 0.01
Add	3.75 ± 0.01	4.08 ± 0.02	4.07 ± 0.01
Triad	3.72 ± 0.04	4.02 ± 0.02	4.00 ± 0.02
RandomAccess (MUP/s)			
MPI_LCG	2.25 ± 0.02	2.64 ± 0.42	34.25 ± 0.30
MPI	2.29 ± 0.02	2.60 ± 0.54	31.58 ± 0.51
Star_LCG	5.71 ± 0.03	13.85 ± 2.17	14.29 ± 0.39
Single_LCG	23.82 ± 1.93	41.92 ± 9.47	47.40 ± 5.16
Star	5.54 ± 0.10	13.23 ± 2.06	13.21 ± 0.20
Single	25.56 ± 2.23	44.71 ± 9.66	46.89 ± 2.46
PTRANS (GB/s)	0.196 ± 0.014	1.181 ± 0.239	1.495 ± 0.019

Table 3: Memory performance metrics for VM, Container, and Host environments. The values are reported as mean ± standard deviation, calculated over three measurements ($n = 3$).

3.2 stress-ng

According to the stress-ng documentation[2], the tool reports a stress test's "throughput" in terms of bogus operations per second (bogo-ops). As highlited in the documentation, it is important to point out that the size and nature of a bogo-op vary depending on the specific stressor being used, making throughput values incomparable across different stressors.

For the CPU stressor, both the host and the container environments demonstrate nearly identical performance. The minor variability observed between them falls well within the expected margin of experimental variation. In contrast, the VM environment exhibits significantly lower throughput – approximately 31% lower than that of the host and container. This degradation in performance is likely attributable to the overhead introduced by the virtualization layer of the hypervisor. A similar trend is observed in the memory stressor

[2] https://wiki.ubuntu.com/Kernel/Reference/stress-ng

Benchmark	VM	Container	Host
Latency (µs)			
MinPingPong	1.957 ± 0.066	1.853 ± 0.056	0.201 ± 0.004
AvgPingPong	49.026 ± 1.597	6.855 ± 0.600	0.211 ± 0.005
MaxPingPong	75.651 ± 2.980	9.962 ± 1.540	0.218 ± 0.004
NaturallyOrderedRing	61.426 ± 7.146	5.926 ± 0.376	0.227 ± 0.005
RandomlyOrderedRing	63.771 ± 8.449	6.872 ± 0.332	0.230 ± 0.003
Bandwidth (GB/s)			
MinPingPong	0.230 ± 0.025	4.995 ± 1.033	11.517 ± 0.725
AvgPingPong	3.617 ± 0.422	7.943 ± 0.674	12.750 ± 0.212
MaxPingPong	11.383 ± 1.381	13.726 ± 0.278	13.908 ± 0.468
NaturallyOrderedRing	0.145 ± 0.020	2.064 ± 0.115	3.101 ± 0.021
RandomlyOrderedRing	0.120 ± 0.014	1.857 ± 0.020	2.971 ± 0.073

Table 4: Latency and Bandwidth Benchmark: VM vs Container vs Host. The values are reported as mean ± standard deviation, calculated over three measurements ($n = 3$).

results. The VM configuration again yields the lowest performance, falling significantly behind both the host and the container environments, which perform comparably to one another.

For this test, two time metrics are of particular interest: Real Time (or wall-clock time), which measures the total elapsed time for the benchmark from start to finish, and Usr+Sys Time (CPU time), which represents the cumulative time the process spent executing in user and system modes. We observe that in general for the CPU stressor, the Real Time and Usr+Sys Time metrics are nearly identical for each environment. This means that the cores are fully utilized during the benchmark, and there is no significant overhead from context switching or other system activities. The system has enough cores (e.g., 4 physical cores for 4 threads) and each thread gets its own core and runs simultaneously. For the memory stressor instead, the Real Time and Usr+Sys Time metrics diverge significantly in the host and container environments. Containers run directly on the host kernel with near-native performance, the divergence in the two times could come from the fact that containers can use memory more efficiently and in full parallel, so each thread contributes more actual CPU time. In this way Usr+Sys time becomes higher than wall-clock time due to summed per-thread CPU time. This asymmetry is notably less pronunced in the VM environment. One possible explanation is that VMs don't access physical memory directly — they go through a virtual memory layer managed by the hypervisor. The net result is that threads are slower, not all the cores are used and less cumulative CPU time is used. The test has been repeated five times for each stressor.

Benchmark	VM	Container	Host
CPU (kBOps/s)			
Real Time	0.924 ± 0.008	1.340 ± 0.013	1.348 ± 0.016
Usr+Sys Time	0.926 ± 0.008	1.342 ± 0.013	1.349 ± 0.016
Memory (kBOps/s)			
Real Time	40.183 ± 0.265	52.180 ± 5.025	53.900 ± 4.948
Usr+Sys Time	41.434 ± 0.952	66.543 ± 2.954	62.687 ± 3.254

Table 5: Stress-ng results for VM, Container, and Host environments. The values are reported as mean ± standard deviation, calculated over five measurements ($n = 5$).

3.3 sysbench

Sysbench provides several test. The key metrics for the CPU benchmarking are Evets/s that measures how many test events the system can handle per second, so the higher the better. And the latency sum that measures the total latency for all executed events, which gives a measure of how long it took to complete all tasks. For the memory benchmark the key metric is the transfer rate, *i.e.*, the memory bandwidth expressed in Gib/s.

The CPU performance are largely consistent across all three environments, with containers achieving slightly better results. For the memory instead we have a clear winner. The container cluster consistently delivers the highest memory bandwidth, outperforming both the VM cluster and the single-host configuration. The VM cluster, in particular, demonstrates a substantial bottleneck in memory performance, indicating inefficiencies in its virtualized memory subsystem. While the single host achieves respectable memory throughput, it does not match the peak performance seen in the container environment. Cumulative latency figures reinforce the efficiency of the container-based setup and highlight the significant memory handling limitations present in the VM cluster. The test has been repeated five times for both CPU and memory.

Benchmark	VM	Container	Host
CPU			
Events/s	453.97 ± 1.28	459.74 ± 2.54	452.38 ± 6.76
Latency sum (ms)	9998.15 ± 0.70	9999.55 ± 0.51	9999.56 ± 0.72
Memory			
Bandwidth (Gib/s)	3.88 ± 0.02	5.51 ± 0.09	5.19 ± 0.08
Latency sum (ms)	1066.09 ± 8.01	839.99 ± 14.63	895.79 ± 18.40

Table 6: Sysbench performance metrics for VM, Container, and Host environments. The values are reported as mean ± standard deviation, calculated over five measurements ($n = 5$).

3.4 IOZone

The Iozone benchmark tests file I/O performance for the following operations: read, write, re-read, re-write, read backwards, read strided, fread, fwrite, random read/write, pread/pwrite variants. Here are reported the the results for the main operations both for vm and containers: read, write, random read, random write. In the following plots (figure 4, 5, 6, 7) we have on the x-axis the file size (KB), on the y-axis the block size (KB) and on the z one the throughput (MB/s). The File size is the size of the file being tested. The transfer size (or Block size) is the size of the I/O operations performed during the test. The Throughput represents how fast data was read during the benchmark using a particular combination of file size and transfer size.

Let's consider for example the results for the **reader** test, Figure 4. Immediately we can note, from the changing in the reference throughput scale (z axis) that there is a significant difference when going from the local file system to the shared one. Moreover, when going from the local to the shared system it is evident that container cluster outperform the VM one. This consideration is in general true also for all the other tests reported below in figure 5, 6, 7.

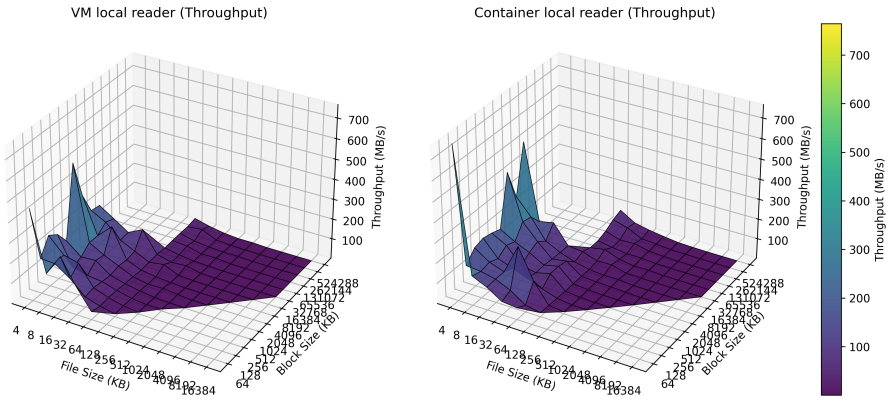


Figure 4: local and shared reader report

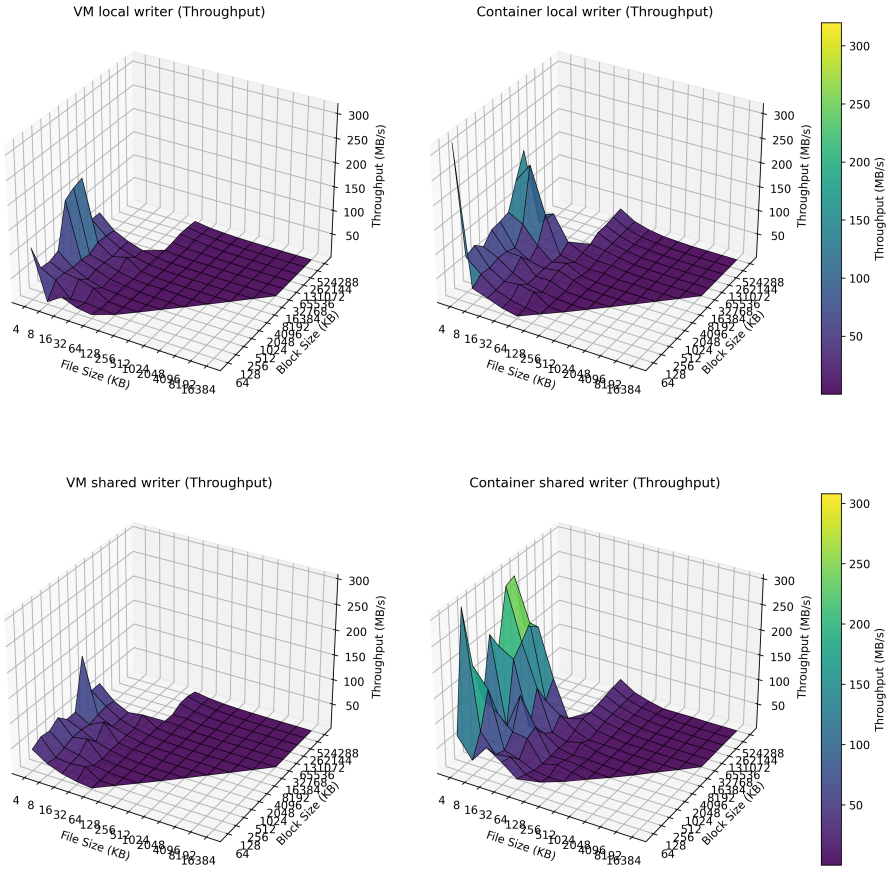


Figure 5: local and shared writer report

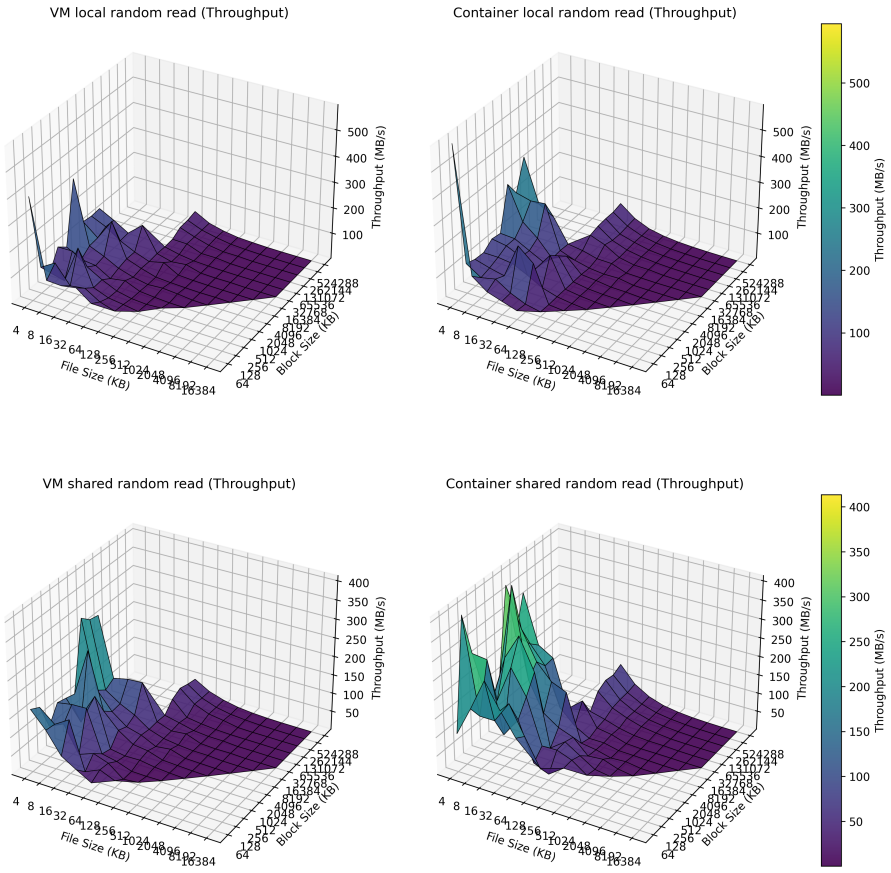
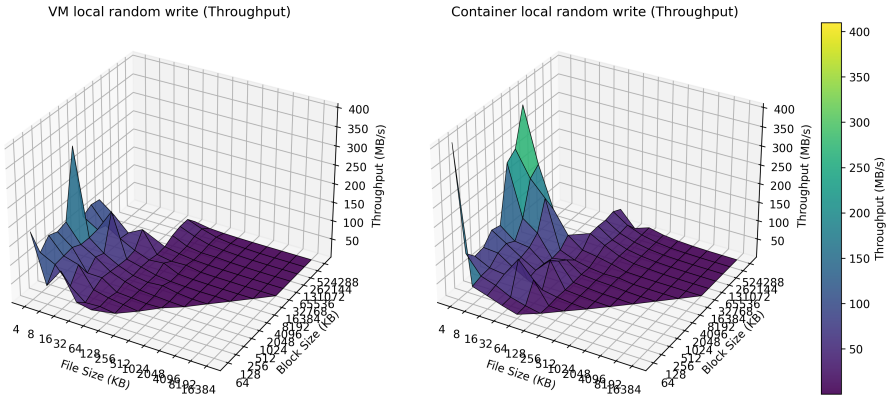


Figure 6: local and shared random read report



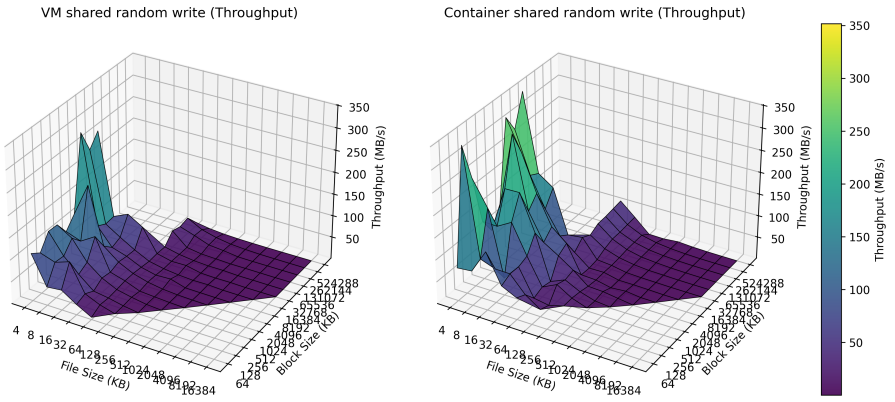


Figure 7: local and shared random write report

To analyze better the I/O performance of the two environment and compare them to the one of the host machine (only for the local filesystem), below is reported a plot of the throughput of the three environments as function of the block size for a fixed file size. In the examples in figure 8, 9, 10, and 11 we can note that the container enviroment, on the shared filesystem, consistently leads to higher throughput than both the VM environment and the host (on the local filesystem). From the plots, we can notice that: the host machine's local filesystem provides a baseline, showing relatively stable performance. Virtualization introduces overhead, which is clearly visible, especially in the VM environment. Docker containers demonstrate significantly better I/O performance than VirtualBox VMs, both on local and shared filesystems. This suggests that the containerization overhead is less impactful on storage performance compared to the full virtualization overhead of VMs in this setup. In VMs, the local filesystem performs noticeably better than the shared filesystem, highlighting the cost of accessing shared storage in this virtualized setup. In Containers, the shared filesystem performs very well, even exceeding the local filesystem at smaller block sizes. This indicates a more efficient shared storage implementation or lower overhead in the Docker environment compared to VirtualBox.

3.5 iperf

Iperf results provide a direct assessment of the network connection quality between nodes in a virtual machine (VM) cluster and those in a Docker container cluster. The Transmission Control Protocol (TCP) benchmarks reveal significantly higher bandwidth in the containerized environment, with both upload and download speeds averaging approximately $42 - 43$ Gb/s. In contrast, the VirtualBox-based VM network achieves substantially lower TCP bandwidth, averaging only $2.7 - 2.8$ Gb/s for both directions. This indicates that the TCP network performance in the container environment is roughly 15 to 16 times greater than that of the VM environment.

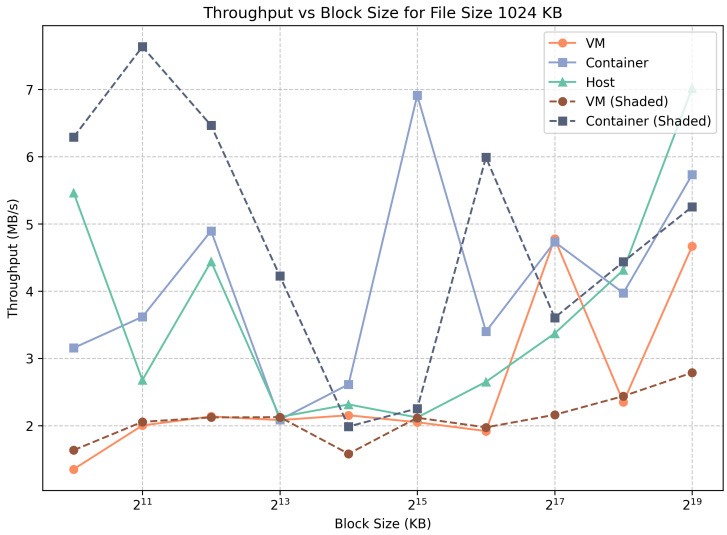


Figure 8: Iozone reader benchmark for a fixed file size of 1024 MB. The throughput is reported in MB/s.

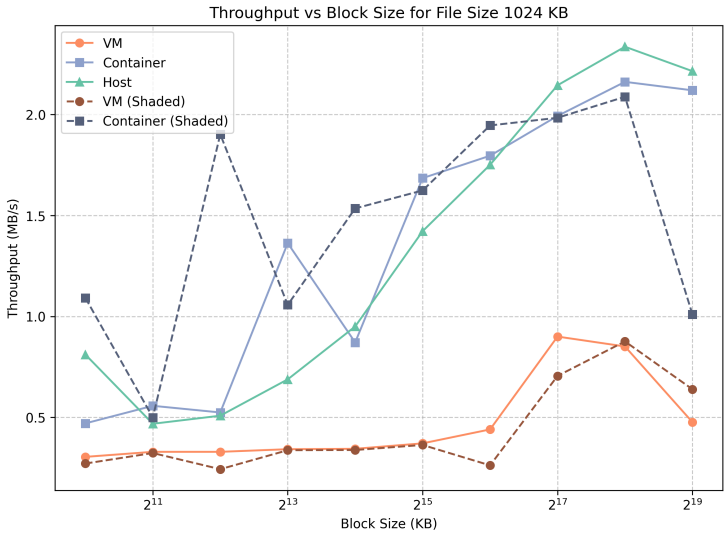


Figure 9: Iozone writer benchmark for a fixed file size of 1024 MB. The throughput is reported in MB/s.

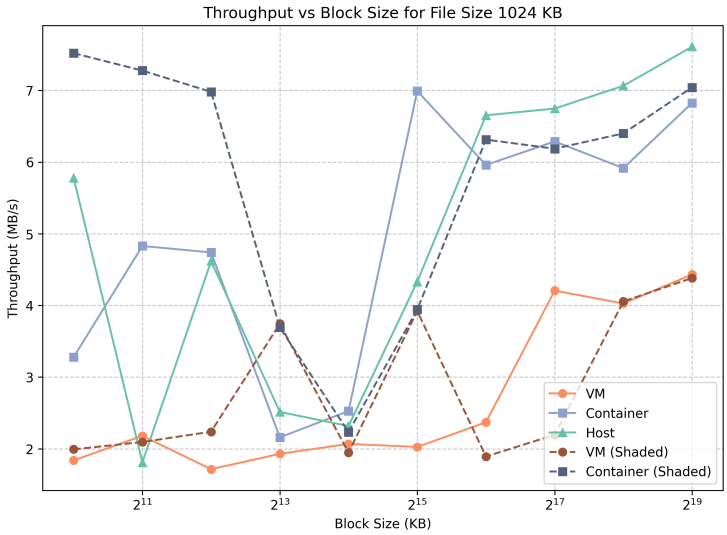


Figure 10: Iozone random read benchmark for a fixed file size of 1024 MB. The throughput is reported in MB/s.

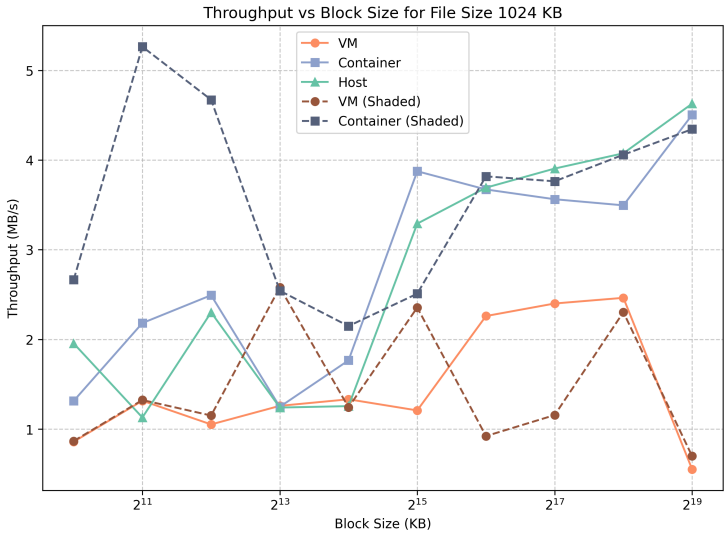


Figure 11: Iozone random write benchmark for a fixed file size of 1024 MB. The throughput is reported in MB/s.

For User Datagram Protocol (UDP) traffic, the successfully received bitrate further underscores the superior performance of the container network, reaching 1.61 Gb/s compared to just 0.44 Gb/s in the VM setup. A critical distinction emerges in the packet loss metrics: the VM network exhibits a high average packet loss rate of 15.85%, meaning that over 15% of UDP packets are dropped even at the relatively modest bitrate of 0.44 Gb/s. Furthermore, the packet loss standard deviation of ±15.99% indicates highly unstable UDP delivery within the VM cluster. Conversely, the container network maintains an exceptionally low packet loss rate of 0.01% at the higher 1.61 Gb/s bitrate, reflecting far more reliable performance.

As observed for the `hpcc` benchmark, the stark contrast in network throughput is a decisive factor in the superior performance of container-based clusters for distributed workloads, such as StarSTREAM or the Message Passing Interface (MPI) components of the HPCC benchmark suite. The constrained network speed and instability in the VM environment inherently limit data transfer rates between nodes, regardless of CPU processing power or local memory performance. Therefore, the network inefficiencies in the VM configuration significantly hinder the execution of communication-intensive parallel applications.

Benchmark	**VM**	**Container**
TCP upload		
Bitrate (Gb/s)	2.66 ± 0.56	42.48 ± 6.75
Transfer (GB)	0.30 ± 0.10	4.91 ± 1.11
TCP download		
Bitrate (Gb/s)	2.84 ± 1.28	43.04 ± 1.66
Transfer (GB)	0.30 ± 0.10	4.97 ± 0.81
UDP		
Bitrate (Gb/s)	0.44 ± 0.19	1.61 ± 0.31
Transfer (GB)	0.05 ± 0.02	0.18 ± 0.06
Lost Datagrams (%)	15.85 ± 15.99	0.01 ± 0.04

Table 7: Iperf results for VM and Container, showing TCP and UDP performance metrics. The values are reported as mean ± standard deviation, calculated over four measurements ($n = 4$).

4 Conclusion

In this project, were implemented and tested two cluster environments: one based on VirtualBox virtual machines and the other using Docker containers. The goal was to evaluate the performance of both environments, particularly in the context of typical benchmarks. The implementation and benchmarking of the two cluster environments highlighted clear differences in both setup

complexity and performance. The Docker-based cluster proved significantly easier to configure, particularly in terms of network setup and shared filesystem integration. Performance testing across CPU, memory, I/O, and network benchmarks consistently showed that the Docker cluster outperformed the VM-based cluster. Notably, the most significant bottlenecks in the virtual machine setup were observed in memory and network performance, which were markedly inferior compared to the containerized environment. These results suggest that container-based clustering offers substantial advantages for lightweight and high-performance deployments in cloud computing contexts.