Final Exam Exercise
Cloud Computing Performance Testing

Giovanni Lucarelli
Università di Trieste

May 8, 2025

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 whole study has been 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 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 assessment tools are described in details in section test-procedures???. 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:
```

```yaml
    # 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 host
    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 --build up -d
```

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 Benchmark**: 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.

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

- `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:

```
node01
node01
node02
node02
```

2.3.1 HPCC

All the tests have been runned in the `/shared` folder, the first step is to locate the `hpcc` executable file and to locate 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_sha
```

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 # 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).

3 Results and Discussion

The full resoults for each test can be found in the folder results in the github repository of the project. Here are briefly discussed only the main ones.

3.1 HPCC

The `hpcc` test has been repeated three times for each environment; here hare reported the mean value (± standard deviation) for the main benchmarks, divided by type of metrics.

3.1.1 Compute Performance

HPL, DGEMM, and FFT are standard benchmarks or computational kernels used to measure the performance of high-performance computing (HPC) systems, especially with respect to floating-point computation and linear algebra workloads. From the results in table **??** we can see that the container consistently leads or ties with host whereas VMs trails behind in most metrics. Note in particular the huge difference in performance of the VMs in the MPIFFT. This benchmark requires frequent communication across MPI ranks — often doing all-to-all exchanges for transposing and redistributing FFT data. Other than the capability in crunching numbers this stresses also: Network latency and bandwidth, MPI transport efficiency, Inter-process communication mechanisms. The critical difference in the three environment is that, whereas the MPI Communication Mechanism of the host is the shared memory, the container uses TCP network protocol but with low overhead and VMs use TCP over virtual Network Interface Controllers (NICs) that returns very poor latency and bandwidth. MPI runs over virtualized network interfaces, causing massive latency. The ping-pong and iperf tests confirm this as shown in the results in table **??** and **??**.

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 ($n = 3$).

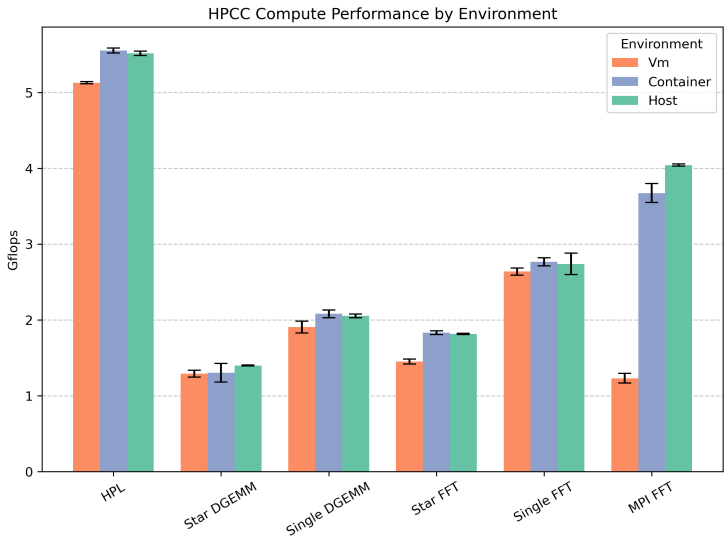


Figure 1: Caption

3.1.2 Memory Benchmarks

The STREAM benchmark provides a measure of the memory bandwidth (GB/s) by testing simple vector operations like Copy, Scale, Add, and Triad. **Single** on a single processor node and **Star** Measures aggregate memory bandwidth across the system using multiple nodes.
Memory Bandwidth (STREAM): Host and container perform similarly, with VM being the weakest. A note on the SingleSTREAM, we can see that Sin-

gleSTREAM reach a good amount of the nominal memory bandwidth in all the environment.
Looking at the nominal performance. By running the command: `sudo dmidecode --type memory`:

```
Configured Memory Speed: 1867 MT/s
Number of memory bank: 2
```

$$\mathrm{BW}_{\mathrm{nom}} = 1.867\frac{\mathrm{GT}}{\mathrm{s}} \times 8\,\frac{\mathrm{B}}{\mathrm{T}} \times 2 = 29.872\frac{\mathrm{GB}}{\mathrm{s}}$$

the Copy benchmark[2] is pretty satisfiyng for all the environments: vm 75%, container 0.81%, host 0.78% ot the nominal bandwidth.
StarSTREAM, running across the two nodes in the VM and Container clusters, is measuring a distributed memory benchmark. This involves significant inter-node communication to exchange data required by the benchmark kernels. The bandwidth in this scenario is not limited by the speed of local memory access but by the speed and latency of the network (or virtual network) connecting the two nodes. The process of sending and receiving data over the network, along with any necessary synchronization and coordination between the nodes, introduces substantial overhead. This overhead drastically reduces the effective memory bandwidth seen by the application, resulting in the much lower figures (around 20% of nominal).
The RandomAccess benchmark measures the rate of random updates to memory (UP/s). It's designed to stress the memory system's ability to handle a large number of small, non-contiguous memory accesses. This is different from STREAM, which focuses on large, sequential data transfers. The MPI Random Access has an interesting behaviour: the Host results are largely out of scale, this can be interpreted thinking at the fact that in vms and containers the mpi protocol is managed through the network adapter and so using the TCP protocol, where as in the host we are in the context of shared memory. MPI processes can use shared memory for inter-process communication, which is orders of magnitude faster than network-based communication This is backed up by the ping pong latency benchmarks. for the vms the latency is much higher, consistent with virtual NIC and TCP stack overhead.
PTRANS (Parallel Matrix Transpose) is designed to measure the global network bandwidth (GB/s) in a parallel computing system. Specifically, it assesses the rate at which large arrays of data can be transferred between the memories of different processors in the system. This shows huge performance difference expecially the VMs compared to the other environment, probably due to the already mentioned network bottleneck.

[2] in particular the Copy benchmark because it is the more optimized by the compiler with an assemply written function!

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 Bandwidth Benchmarks: VM vs Container vs Host (GB/s)

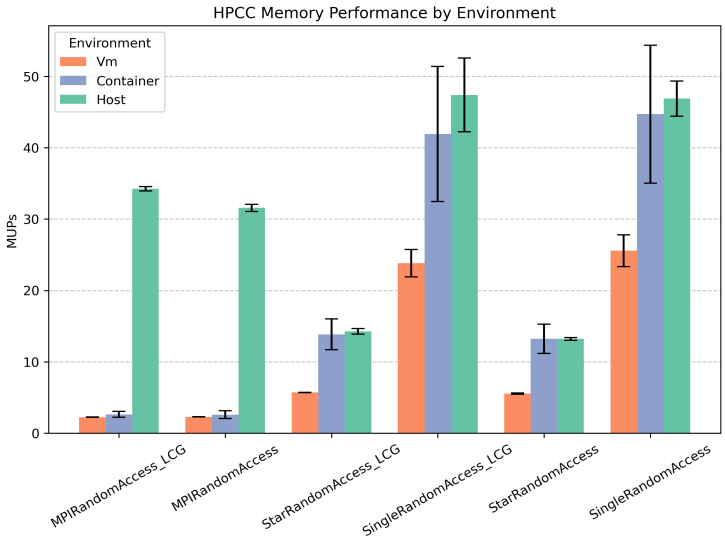


Figure 2: Caption

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

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 Benchmarks: VM vs Container vs Host

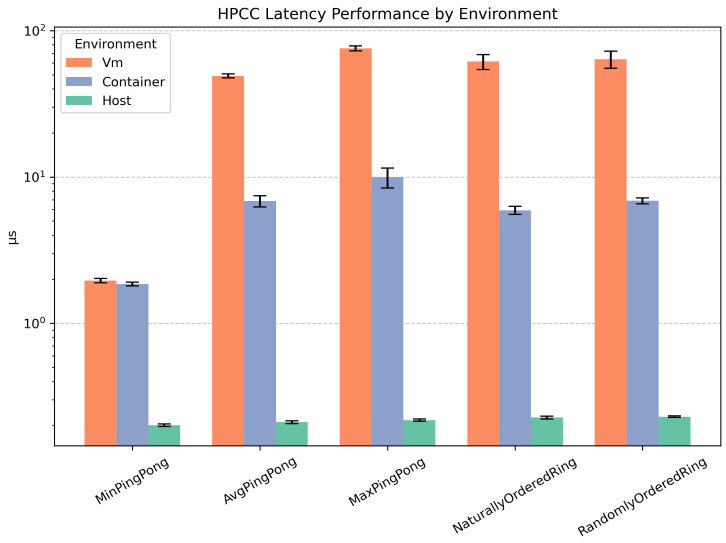


Figure 3: Note the log scale

3.2 stress-ng

From the stress-ng documentation[3]: " Stress-ng measures a stress test "throughput" using "bogus operations per second". The size of a bogo op depends on the stressor being run, and are not comparable between different stressors."
Regarding the cpu stressor, host and container shows nearly identical performances, the light variability is negligible and is inside the variability of the tests. VMs throughput performances are significantly lower, about the 31% lower than host/container. This is probably due to the virtual overhead introduced by the hypervisor.
Similiarly to the cpu, for the memory stressor the VM results are again the lowest, significantly lower than host and container, again similiar. Real Time (Wall-Clock Time) is the total time that elapsed from when the benchmark started until it finished, as measured by a clock on the wall, Usr+Sys Time (CPU Time) is the total amount of CPU time that the benchmark process spent executing code. The large difference between Real Time and Usr+Sys Time for the Container and Host in the Memory VM stressor is the expected result of running a multi-threaded benchmark effectively utilizing multiple CPU cores. The absence of this large difference in the VM environment highlights that the virtualization layer in VirtualBox introduces overhead or inefficiencies that hinder the effective parallel execution and CPU utilization of this specific memory-intensive multi-threaded workload compared to running it directly on the host or within a Docker container.

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: kilo BogoOps/s for each benchmark, number of repetition of the experiment $n = 5$

3.3 sysbench

CPU Performance: The CPU performance per node is largely uniform across all three environments in terms of average metrics. The addition of maximum latency shows that the single Host is slightly more stable in avoiding high-latency CPU events compared to the clustered environments (VM and Container). The total latency sum is dominated by the test duration. For typical CPU workloads, there's no significant advantage or disadvantage to using VMs or Containers in a cluster compared to a single host based on these results. Memory Performance:

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

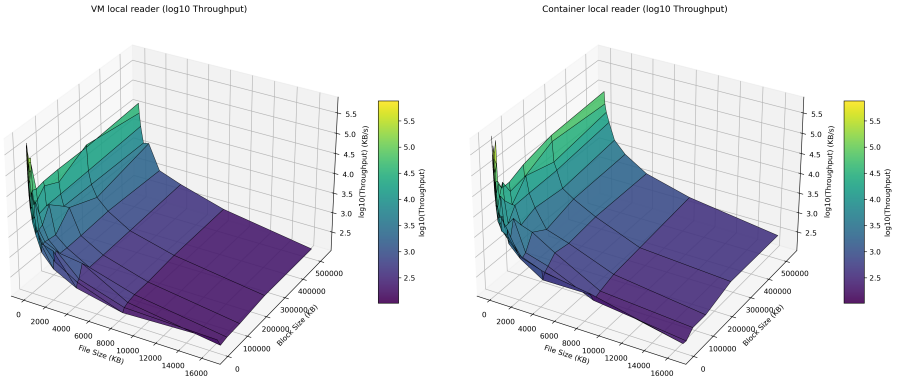
This is where the significant differences lie. The Container cluster setup provides superior memory throughput per node compared to both the VM cluster and the single Host. The VM cluster exhibits a considerable performance bottleneck for memory operations. The single Host is a solid performer but doesn't reach the peak memory throughput of the Container node. The maximum latency metric suggests slightly less predictability in memory access times within the clustered environments compared to the single host, although the differences are relatively small. The sum of latencies strongly supports the Container's efficiency and the VM's inefficiency in memory handling.

Benchmark	**VM**	**Container**	**Host**
CPU			
Events/s	453.97 ± 1.28	459.74 ± 2.54	452.38 ± 6.76
Latency min (ms)	2.04 ± 0.01	2.03 ± 0.00	2.05 ± 0.02
Latency avg (ms)	2.20 ± 0.01	2.17 ± 0.01	2.21 ± 0.03
Latency max (ms)	6.05 ± 1.18	6.14 ± 2.73	5.49 ± 1.56
Latency sum (ms)	9998.15 ± 0.70	9999.55 ± 0.51	9999.56 ± 0.72
Memory			
Transfer rate (Gib/s)	3.88 ± 0.02	5.51 ± 0.09	5.19 ± 0.08
Latency max (ms)	1.26 ± 0.50	1.32 ± 0.62	1.09 ± 0.24
Latency sum (ms)	1066.09 ± 8.01	839.99 ± 14.63	895.79 ± 18.40

Table 6: Caption

3.4 IOZone

For clarity the plots of the host are not reported(?) but can be found here.



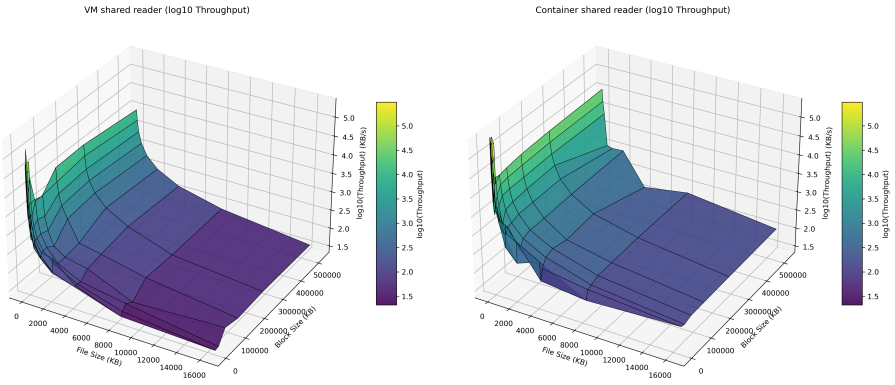


Figure 4: local and shared reader report

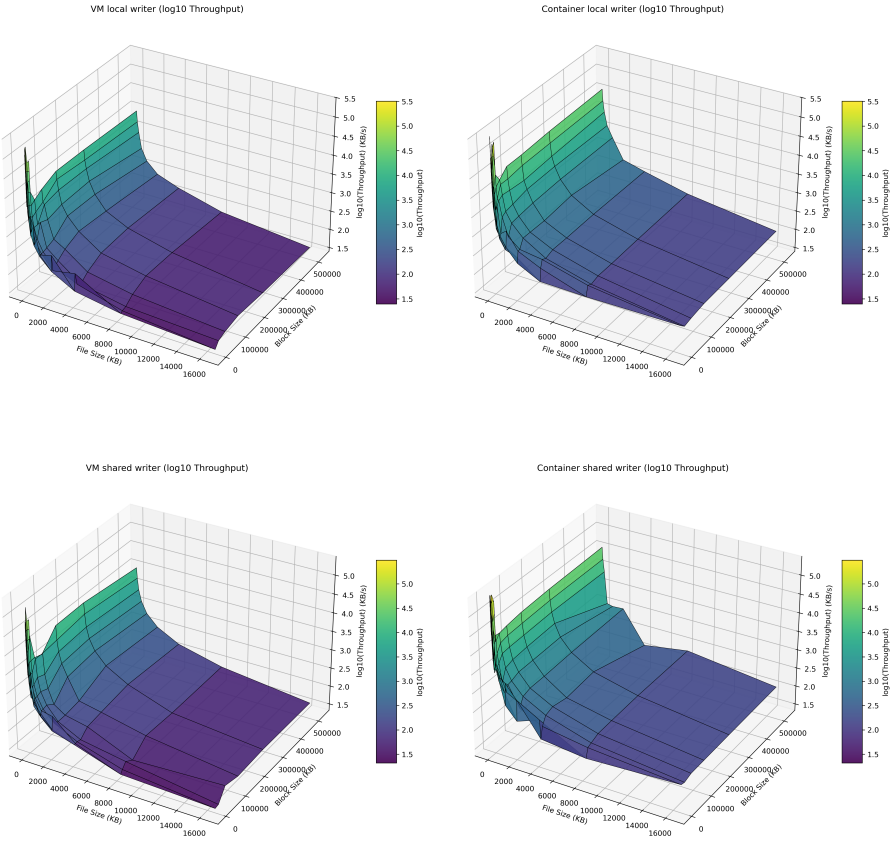


Figure 5: local and shared writer report

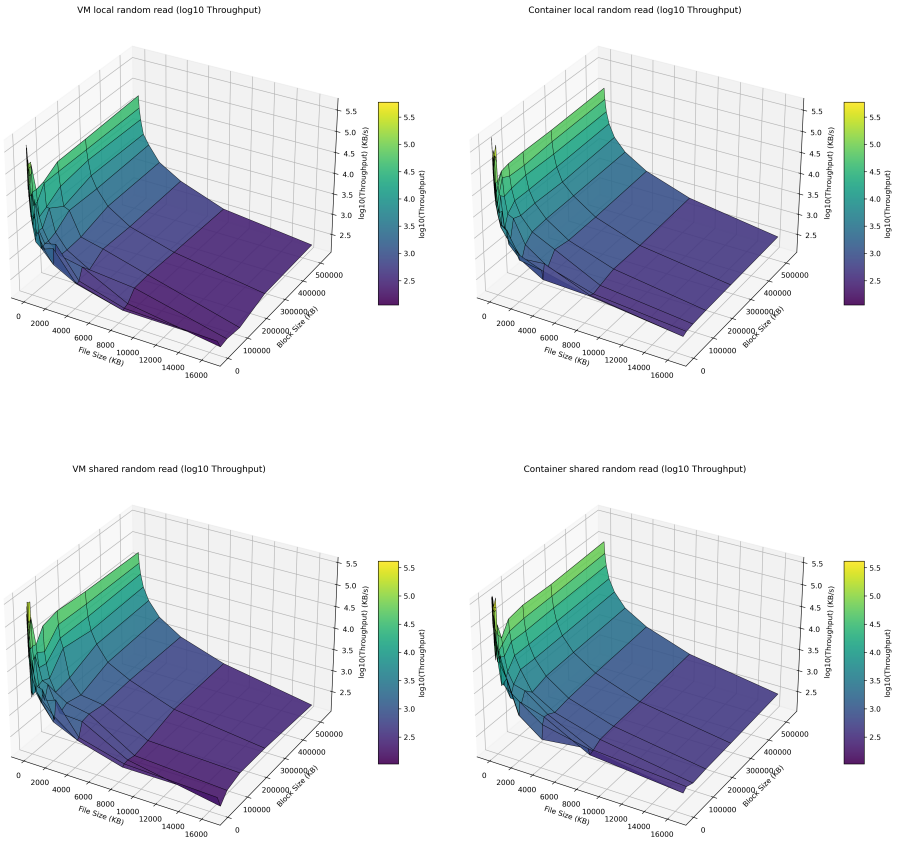
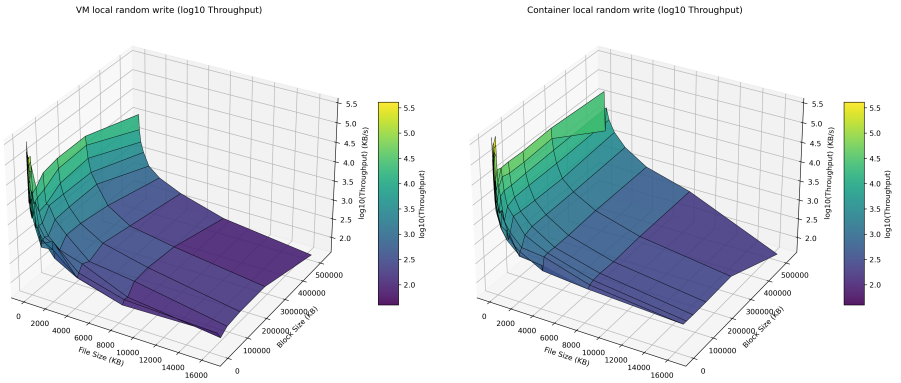


Figure 6: local and shared random read report



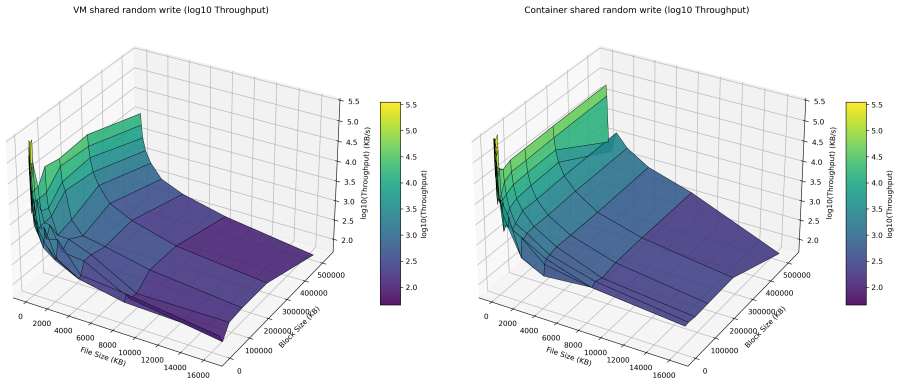


Figure 7: local and shared random write report

3.5 iperf

iperf is a standard tool used to measure network performance, specifically throughput (bandwidth), latency, and jitter. These results directly assess the quality of the network connection between the nodes in your VM cluster versus the nodes in the Container cluster.

TCP Performance (Upload and Download): The bandwidth (Bitrate) is the most striking difference. The Container environment achieves a TCP bitrate of approximately 4.9 Gb/s (both upload and download), while the VM environment is limited to a mere 0.30 Gb/s. This means the network link between the two nodes in the Docker cluster is about 16 to 17 times faster for sustained TCP data transfer than the virtualized network link between the two nodes in the VirtualBox cluster. The Containers are effectively utilizing a much higher bandwidth pipe than the VMs. The relative standard deviations (±) show that the container network performance is not only much higher but also more consistent than the VM network performance.

The iperf results provide the crucial evidence to explain the performance differences observed in benchmarks that rely on inter-node communication. The iperf results definitively confirm the statement that "VMs rely on virtualized NICs with limited performance." The measured TCP bandwidth of 0.30 Gb/s is a severe bottleneck for any distributed application or benchmark (like MPI in HPCC or StarSTREAM) that attempts to send data between the VM nodes. This low network speed directly limits how fast data can be moved between nodes, regardless of how fast the CPUs can process it or how fast local memory is.

The much higher network bandwidth (4.9 Gb/s) and lower jitter in the Container environment explain why the Container cluster would outperform the VM cluster on distributed benchmarks. With a faster and more stable network, the overhead of inter-node communication is significantly reduced, allowing the nodes to exchange data more efficiently and complete the distributed tasks (like

those in StarSTREAM or HPCC's MPI components) faster.

Benchmark	**VM**	**Container**
TCP upload		
Transfer (GB)	2.66 ± 0.56	42.48 ± 6.75
Bitrate (Gb/s)	0.30 ± 0.10	4.91 ± 1.11
TCP download		
Transfer (GB)	2.84 ± 1.28	43.04 ± 1.66
Bitrate (Gb/s)	0.30 ± 0.10	4.97 ± 0.81
UDP		
Transfer (MB)	0.12 ± 0.02	0.13 ± 0.00
Bitrate (Mb/s)	1.04 ± 0.08	1.05 ± 0.00
Jitter (ms)	1.60 ± 0.74	0.15 ± 0.08

Table 7: Caption

4 Conclusion

In general containers are better than VMs, and results from this analysis show that they are significantly better, in particular, they provide a better alternative expecially with respect to memory, I/O, and connection.