Tavano Matteo SM3800057

Cloud Basic Assignment – Performance Testing

Introduction:

This project focuses on the performance evaluation of cloud infrastructure, specifically assessing the deployment of **Virtual Machines (VMs)** using the Oracle VM VirtualBox hypervisor and containerization with **Docker**. The performance tests conducted in both environments aim to analyze key **metrics**, including CPU utilization, memory consumption, disk read/write operations, and network communication between nodes within a subnet.

The primary goal of this project is to conduct a comprehensive performance assessment of cloud-based infrastructures by deploying Virtual Machines (VMs) via the Oracle VM VirtualBox hypervisor and containerized environments using Docker. The evaluation aims to **benchmark** and compare key performance indicators. This analysis will provide insights into the computational overhead, resource efficiency, and overall scalability of both virtualization and containerization approaches in cloud computing environments.

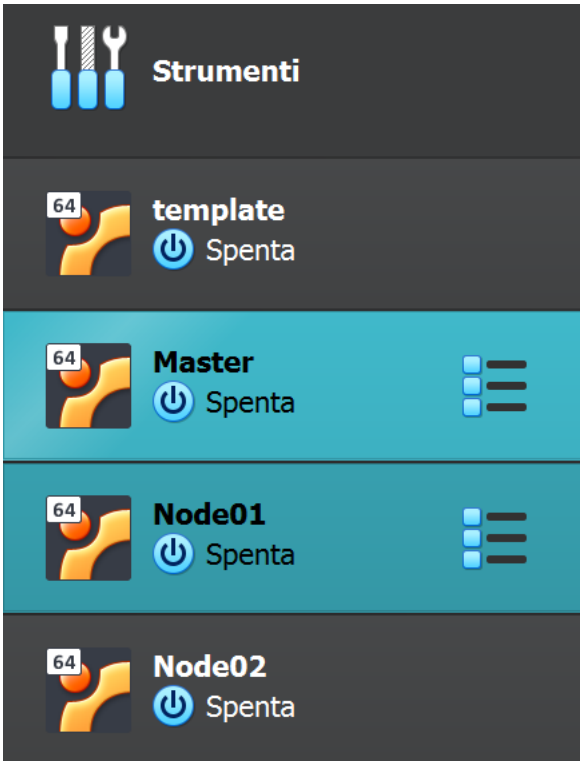
Part I: Virtual Machines Performance Test

Setup:

To facilitate the performance evaluation, a virtualization environment was established using Oracle VM VirtualBox as the chosen hypervisor. A baseline virtual machine, referred to as *Template*, was created with a configuration of 1 vCPU, 1 GB of RAM, and a 25 GB virtual hard disk. The operating system selected for deployment was Ubuntu 24.04 LTS Server.

https://ubuntu.com/download/server.

To enable internal **network communication** between virtual machines, the *Template* was cloned into three distinct instances: *Master*, *Node01*, and *Node02*. Each of these virtual machines was scaled up 2 vCPUs, 2 GB of RAM, and maintained a fixed system configuration to ensure consistency in performance testing. This uniform allocation of resources across the VMs is a critical factor in obtaining reliable benchmarking results.



After cloning the three virtual machines, specific network configurations were modified within Oracle VM VirtualBox to facilitate connectivity. Each VM was configured with two network interfaces: the first interface (*Scheda1*) was set to NAT mode to enable external connectivity, while the second interface (*Scheda2*) was assigned to an Internal Network named *clustervimnet*. This configuration establishes a shared network environment, allowing the virtual machines to communicate using local IP addresses within the cluster.

Install Dependencies:

- sudo apt update
- sudo apt upgrade -y
- sudo apt install -y openssh-server net-tools gcc make build-essential gfortran openmpi-bin openmpi-common libopenmpi-dev
- sudo systemctl enable --now ssh
- sudo systemctl start ssh

To ensure effective communication within the cluster, the network configuration of each virtual machine was modified. This required editing the *50-cloud-init.yaml* file located in the */etc/netplan/* directory. The configuration file was updated as follows:

- sudo nano /etc/netplan/50-cloud-init.yaml

Next in this report, it is set another possible configuration.

Interface enp0s8: Assigned a static IP address for each node within the *clustervimnet* internal network:

- *Master*: 192.168.0.1
- *Node01*: 192.168.0.2
- *Node02*: 192.168.0.3

Interface enp0s3: Configured to use DHCP (dhcp4: true) to enable NAT-based external connectivity.

The configuration changes were applied using the following command to ensure proper network functionality:

- sudo netplan apply

To validate the network configuration (details of which will be discussed later), connectivity between virtual machines can be tested using the *ping* command. For example, from the *Master* node, the following command can be executed:

- ping -c 4 192.168.0.2

If the setup is correctly configured, *Node01* should receive and respond to the packets sent from the *Master* node, confirming successful communication within the internal network.

1. Configure Master Node

Set the host name:

To ensure correct node identification, set the hostname of the master node:

- Sudo hostname set-hostname master

Edit the hosts file:

The /etc/hosts file needs to be edited to define static IP mappings for all nodes:

- Sudo nano /etc/hosts

```
127.0.0.1 localhost
127.0.1.1 master
192.168.0.1 master
192.168.0.2 node01
192.168.0.3 node02

# The following lines are desirable for IPv6 capable hosts
::1 ip6-localhost ip6-loopback
fe00::0 ip6-localnet
ff00::0 ip6-mcastprefix
ff02::1 ip6-allnodes
ff02::2 ip6-allrouters
```

Install and Configure Dnsmasq

To provide DHCP and DNS services for the cluster:

- sudo apt install -y dnsmasq

Edit the Dnsmasq configuration file:

- sudo nano /etc/dnsmasq.conf

```
interface=enp0s8
bind-interfaces
dhcp-range=192.168.0.2,192.168.0.254,255.255.255.0,12h
dhcp-option=3,192.168.0.1
dhcp-option=6,192.168.0.1
bogus-priv
cache-size=1000
resolv-file=/etc/resolv.dnsmasq
listen-address=127.0.0.1, 192.168.0.1
log-queries

# Static IP assignments for nodes using their MAC addresses
dhcp-host=mac::address::node02,node01,192.168.0.2
dhcp-host=mac::address::node02,node02,192.168.0.3
```

Configure DNS Resolution:

Create a new resolv.conf file with the correct DNS settings:

- sudo nano /etc/resolv.conf

```
nameserver 127.0.0.1
options edns0 trust-ad
search .
```

Enable External Access and Reboot:

To ensure external access is available:

- sudo ln -s /run/systemd/resolve/resolv.conf /etc/resolv.dnsmasq
- sudo systemctl restart dnsmasq systemd-resolved
- sudo systemctl enable dnsmasq
- sudo reboot

2. Setup Distributed File System

Install NFS Server on Master Node:

- sudo apt install nfs-kernel-server

Create a shared directory accessible to all nodes:

- sudo mkdir /shared
- sudo chmod 777 /shared

Configure NFS Exports:

Edit NFS configuration file:

- sudo nano /etc/exports

Add the entry:

- /shared/ 192.168.0.0/255.255.255.0(rw,sync,no_root_squash,no_subtree_check)

Restart NFS Server:

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

3. Configure and compute the nodes (Node01 – Node02)

Clone as mentioned before (use OracleVM setup menu)

Install Dependencies:

- sudo apt update
- sudo apt upgrade -y
- sudo apt install -y openssh-server net-tools libblas3 libblas-dev libopenmpi-dev openmpi-bin openmpi-common

Enable and start SSH:

- sudo systemctl enable –now ssh
- sudo systemctl start ssh

Configure network setting for nodes:

As for master node:

- sudo nano /etc/netplan/50-cloud-init.yaml

```
network:
  ethernets:
    enp0s3:
      dhcp4: true
      dhcp4-overrides:
        use-dns: false
    enp0s8:
      dhcp4: true
      dhcp6: false
      dhcp-identifier: mac
  version: 2
```

- sudo netplan apply

Set Hostnames for Node01

- sudo hostname set-hostname node01

Configure DNS for the Node:

Remove the existing resolv.conf symlink:

- sudo unlink /etc/resolv.conf

Create a new file:

- sudo nano /etc/resolv.conf

Add:

```
nameserver 192.168.0.1
search .
```

Edit the host file for Node01:

- sudo nano /etc/hosts

```
127.0.0.1 localhost
127.0.1.1 node01
192.168.0.1 master
192.168.0.2 node01
192.168.0.3 node02
```

Repeat the steps for Node02;

4. Enable Passwordless SSH Access

From the Master node generate a ssh key:

- ssh-keygen -t rsa

Copy to both nodes:

- ssh-copy-id user01@node01
- ssh-copy-id user01@node02

Now the master can access the nodes via SSH without requiring a password, facilitating distributed HPCC execution.

CPU Test HPC Challenge Benchmark:

In this section, the **HPC Challenge (HPCC) Benchmark** will be utilized to assess high-performance computing capabilities on Ubuntu-based systems. HPCC is a comprehensive benchmarking suite that evaluates processor performance, memory subsystem efficiency, and interconnect bandwidth. It includes several tests, such as HPL, STREAM, DGEMM, PTRANS, RandomAccess, and FFT, each targeting different aspects of system performance. http://icl.cs.utk.edu/hpcc/

Key optimizationin **High-Performance Linpack (HPL)** benchmark parameters include *block size* (NB), which depends on the CPU architecture, as well as grid dimensions *(P and Q*) that should match the number of MPI processes. The problem *size (N)* is another crucial factor in performance tuning.

Downloading HPCC and Preparing the Environment: (version hpcc-1.5.0)

The HPCC benchmark suite needs to be downloaded and extracted in a shared directory to ensure accessibility across multiple nodes. Using a shared folder allows all nodes within the cluster to access the benchmark without redundant copies. The shared directory is mounted as /shared on all nodes.

- cd /shared
- wget https://icl.utk.edu/projectsfiles/hpcc/download/hpcc-1.5.0.tar.gz
- tar xzf hpcc-1.5.0.tar.gz
- cd hpcc-1.5.0

This sequence downloads the HPCC version 1.5.0 archive, extracts its contents, and navigates into the extracted directory.

Configuring the Build System:

The compilation process of HPCC requires an appropriate makefile. A template configuration file **(Make.Linux_PII_CBLAS)** is selected as the basis due to its compatibility with CBLAS and x86 architecture. The following command copies the template configuration file to the appropriate location:

- cp hpl/setup/Make.Linux_PII_CBLAS hpl/Make.Linux

Editing Make.Linux for the Correct Library Paths

The Make.Linux file must be modified to correctly specify paths for compilation and linking. This step ensures that the HPCC build system correctly locates necessary dependencies.

- nano hpl/Make.Linux

Makefile variables:

- **TOPdir**: Set to /shared/hpcc-1.5.0, ensuring all relative paths within the makefile correctly reference the shared HPCC directory.
- **MPdir**: Set to /, as OpenMPI's system-wide installation in Ubuntu places relevant libraries and headers under /usr.
- **MPinc (MPI Include Paths)**: Extended to include OpenMPI headers along with HPCC and HPL-specific directories, ensuring correct compilation.
- **MPlib (MPI Library Path)**: Adjusted to -L/usr/lib/x86_64-linux-gnu -lmpi, linking against OpenMPI's dynamically linked libraries instead of static ones.
- **LAdir (Linear Algebra Directory)**: Modified to /usr/lib/x86_64-linux-gnu, which contains the system-installed BLAS and LAPACK implementations.
- **LAlib (Linear Algebra Library)**: Changed to -lblas, linking directly to the system-provided BLAS library rather than a custom-compiled one.
- **Fortran Compatibility (F2CDEFS)**: Defined as -DAdd_ -DF77_INTEGER=int -DStringSunStyle, ensuring proper name mangling and integer type consistency between Fortran and C.
- **LINKER**: Set to $(CC), ensuring a consistent compiler toolchain is used.
- **RANLIB**: Updated to ranlib to utilize standard library indexing.

Compiling HPCC:

Once the Make.Linux file is correctly configured, it is copied to the HPCC root directory to be used during the build process:

- cd /shared/hpcc-1.5.0
- cp hpl/Make.Linux Make.Linux

Call makefile:

- make arch=Linux

This command builds the HPCC executable, compiling the benchmark suite with the specified settings.

Configuring the Hostfile for MPI Execution

To enable multi-node execution, a hostfile is created that specifies the participating nodes. This file informs OpenMPI of the available compute nodes and their resource allocation.

- Sudo nano /shared/hpcc-1.5.0/hostfile

The following entries are added:

```
node01 slots=1
node02 slots=1
```

Each node is assigned one processing slot, allowing OpenMPI to distribute processes accordingly.

Mounting the Shared Folder on Compute Nodes

Before executing HPCC, the shared folder containing the benchmark must be accessible on all nodes. This is achieved by mounting the shared directory from the master node:

- Sudo mount master:/shared /shared

This ensures that the executable and required files are available across all participating nodes.

Launching the HPCC Benchmark

The HPCC benchmark is executed using OpenMPI with the following command:

- /usr/bin/mpirun -np 2 --display-map --map-by node --hostfile hostfile \ --mca btl tcp,self --mca btl_tcp_if_include enp0s8 -v ./hpcc > hpcc_summary.txt 2>&1

The execution involves several key configurations:

- **Process Distribution**: The -np 2 option specifies two parallel processes, which are distributed across nodes as dictated by --map-by node.
- **Network Configuration**: The --mca btl tcp,self flag enforces the use of the TCP transport layer for inter-node communication. The --mca btl_tcp_if_include enp0s8 option ensures communication occurs over the specified network interface (enp0s8).
- **Process Mapping**: The --display-map flag displays how processes are mapped across nodes, aiding in debugging and performance analysis.
- **Hostfile Utilization**: The --hostfile hostfile argument ensures OpenMPI distributes processes according to the predefined node allocation.
- **Verbose Output**: The -v flag enables detailed logging.
- **Output Redirection**: Standard output and error messages are captured in hpcc_output.txt for later inspection.

Upon completion, the output file can be analyzed to assess the performance of the benchmark, providing insights into computational efficiency and inter-node communication latency.

Comments on the result file (hpcc_summary.txt)

The performance benchmark results for the HPCC test suite on the virtualized cluster indicate modest computational capabilities with some variability across different metrics. The HPL benchmark shows a peak performance of *around 0.0077 TFlops*, which is relatively low, likely due to resource constraints imposed by the virtualization environment and limited *CPU and memory per node*. *STREAM* memory bandwidth results vary significantly, with some measurements reaching over **100 GB/s in** smaller vector sizes, but overall performance remains inconsistent, suggesting memory access bottlenecks. *DGEMM* results hover around 3–4 GFlops, reflecting the limited floating-point processing power available. *PTRANS* network bandwidth is moderate, peaking at approximately 6 GB/s in optimal conditions, but generally remains below **1 GB/s**, highlighting inter-node communication limitations. *RandomAccess* performance is notably low, with GUPs values well below expectations, indicating significant latency in random memory access operations. Overall, the benchmark results suggest that while the cluster can handle basic distributed computations, performance is constrained by virtualization overhead, network bandwidth, and memory access efficiency.

General System Test:

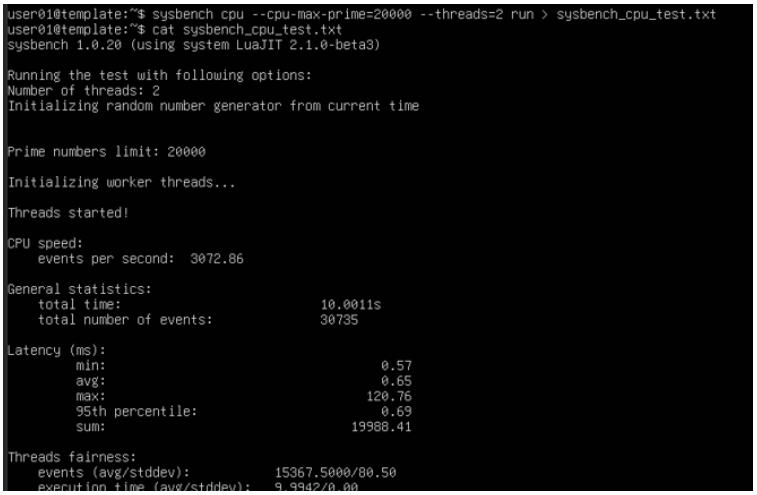
After completing the HPL benchmark, a general system performance test was conducted using the *stress-ng* and *sysbench* utilities. These tests were executed on the *Master* node and *Node01* to assess system stability and resource utilization. The results, along with the command-line execution and parameter configurations, will be analyzed in the following sections.For this, stress-ng and sysbench has been downloaded.

- sudo apt update && sudo apt install stess-ng sysbench -y

The subsequent sections will present a summary of the results and include relevant screenshots captured during the testing process.

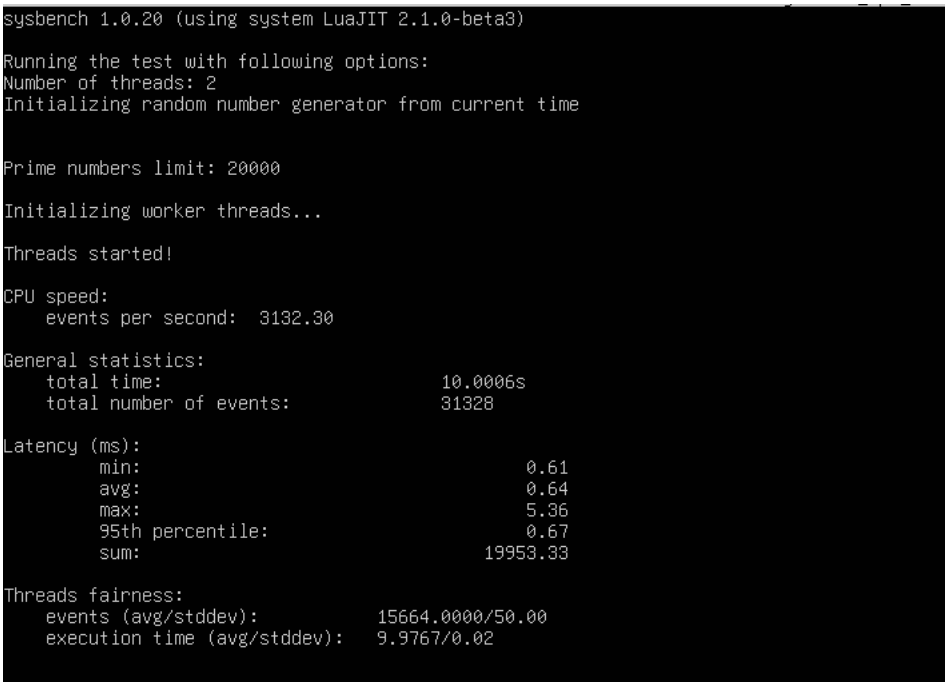
<u>CPU TEST (SYSBENCH) on Master:</u>

- sysbench cpu --cpu-max-prime=20000 --threads=2 run



CPU TEST (SYSBENCH) on Node:

- sysbench cpu --cpu-max-prime=20000 --threads=2 run



Comments on the results:

Metrics:

- **Total number of events**: The number of prime numbers calculated during the test.
- **Latency (ms):** Measures the response time of computations, with average, minimum, and maximum values reported. Lower values indicate better performance.
- **CPU events per second:** The number of completed operations per second, representing CPU efficiency.
- **Execution time:** The total duration of the test, including CPU and system overhead.

Comparison:

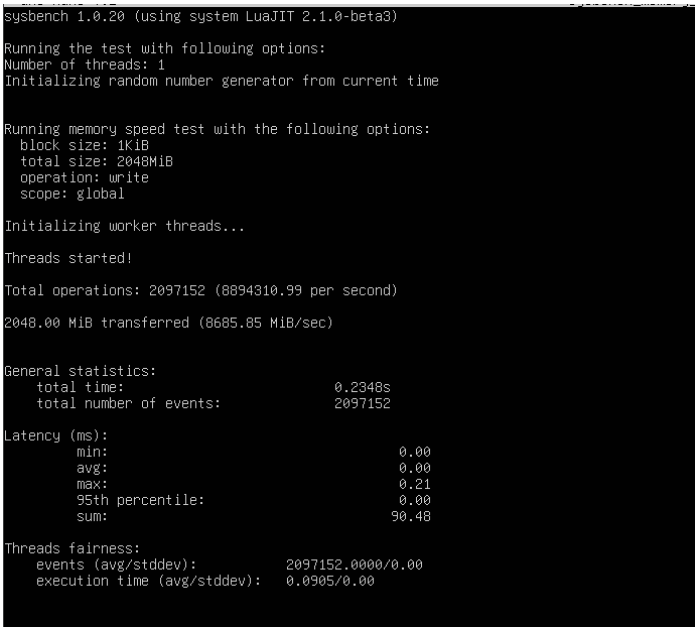
- Both nodes ran the test with **two threads** and a **maximum prime number of 20,000**, ensuring a direct comparison.
- The **Master node** completed **slightly fewer CPU events per second** compared to **Node01**, suggesting minor variations in performance.
- **Latency values** are similar, indicating comparable processing efficiency.
- The **execution time is slightly different**, possibly due to background processes or slight hardware variations.

Conclusions:

The results show that both nodes exhibit similar computational capabilities, with minor differences in CPU performance. These variations could result from differences in system load or scheduling during the test execution.

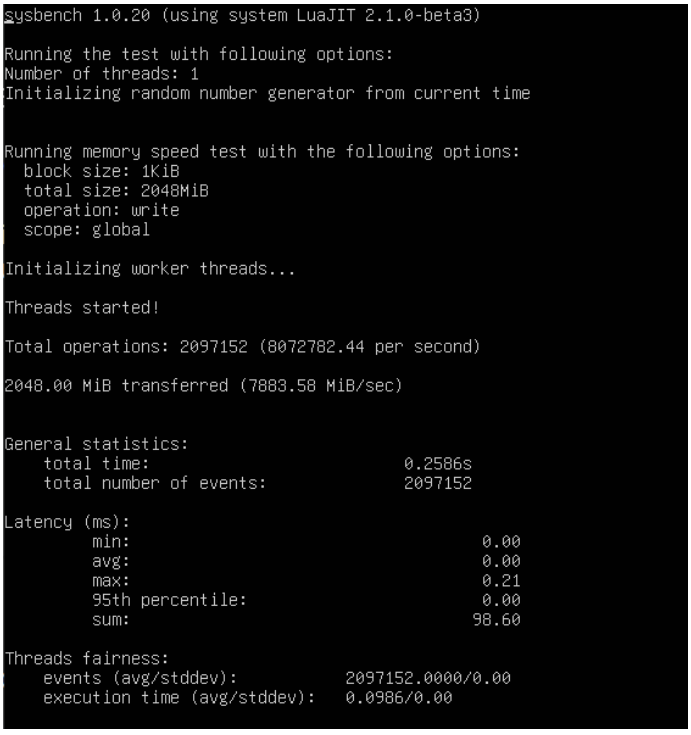
MEMORY TEST (SYSBENCH) on Master:

- sysbench memory –memory-total-size=2G run



MEMORY TEST (SYSBENCH) on Node:

- sysbench memory –memory-total-size=2G run



Comments on the results:

Metrics:

- **Total operations performed**: The number of memory operations executed during the test.
- **Memory transfer rate (MB/sec)**: Measures how fast data is transferred in memory. Higher values indicate better memory performance.
- **Latency (ms)**: The response time of memory operations, with average, minimum, and maximum values reported.
- **Execution time**: The total duration of the test.

Comparison:

- Both nodes performed the test with a **total allocated memory size of 2GB**, ensuring consistency.
- The **total number of operations and transfer rate** appear **similar** between the Master and Node01, indicating comparable memory bandwidth and efficiency.
- **Latency values** show only minor variations, suggesting similar memory response times.
- **Execution time is nearly identical**, confirming that both nodes exhibit comparable memory performance.

Conclusions:

The memory performance results show no significant differences between the Master and Node01, indicating consistent memory bandwidth and efficiency across both nodes. The minor variations observed could be attributed to system load fluctuations during testing.

CPU TEST (STRESS-NG) on Master:

- stress-ng –cpu 2 –cpu-method matrixprod –timeout 60s –metrics-brief

```
stress-ng: info:  [19092] setting to a 1 min, 0 secs run per stressor
stress-ng: info:  [19092] dispatching hogs: 2 cpu
stress-ng: info:  [19092] note: /proc/sys/kernel/sched_autogroup_enabled is 1 and this can impact scheduling throughput for processes not attached to a tty.
stress-ng: metrc: [19092] stressor       bogo ops real time  usr time  sys time   bogo ops/s     bogo ops/s
stress-ng: metrc: [19092]                          (secs)    (secs)    (secs)   (real time) (usr+sys time)
stress-ng: metrc: [19092] cpu              372744     60.00    116.49      3.43      6212.25        3108.30
stress-ng: info:  [19092] skipped: 0
stress-ng: info:  [19092] passed: 2: cpu (2)
stress-ng: info:  [19092] failed: 0
stress-ng: info:  [19092] metrics untrustworthy: 0
stress-ng: info:  [19092] successful run completed in 1 min, 0.03 secs
```

CPU TEST (STRESS-NG) on Node:

- stress-ng –cpu 2 –cpu-method matrixprod –timeout 60s –metrics-brief

```
stress-ng: info:  [2558] setting to a 1 min, 0 secs run per stressor
stress-ng: info:  [2558] dispatching hogs: 2 cpu
stress-ng: info:  [2558] note: /proc/sys/kernel/sched_autogroup_enabled is 1 and this can impact scheduling throughput for processes not attached to a tty.
stress-ng: metrc: [2558] stressor       bogo ops real time  usr time  sys time   bogo ops/s     bogo ops/s
stress-ng: metrc: [2558]                          (secs)    (secs)    (secs)   (real time) (usr+sys time)
stress-ng: metrc: [2558] cpu              133820     60.75    108.33     12.55      2202.64        1107.02
stress-ng: info:  [2558] skipped: 0
stress-ng: info:  [2558] passed: 2: cpu (2)
stress-ng: info:  [2558] failed: 0
stress-ng: info:  [2558] metrics untrustworthy: 0
stress-ng: info:  [2558] successful run completed in 1 min, 0.78 secs
```

Comments on the results:

Metrics:

- **CPU operations (bogo ops)**: Measures the number of "bogus operations" performed as a synthetic workload, useful for relative comparisons.
- **Real time, user time, and system time (secs)**: *Real time* is the total duration of the test, *User time* is the CPU time spent executing user-space processes and *System time* is the CPU time spent executing kernel operations.
- **Bogo ops per second**: A relative measure of computational throughput. Higher values indicate better performance.

Comparison:

- Both tests were run using **two CPU threads** and the *matrixprod* method for **60 seconds**, ensuring a direct comparison.
- The **Master node** achieved a **higher number of total CPU operations (327,474 vs. 133,892)**, indicating significantly better computational throughput.
- The **bogo ops per second** on the Master (6,212.25) is **almost three times higher** than on Node01 (2,202.64), suggesting a performance gap.
- **User and system times** on the Master are also notably higher, aligning with its increased workload processing efficiency.

Conclusions:

The Master node demonstrated **substantially better CPU performance** compared to Node01 under the same test conditions. The discrepancy could be attributed to differences in background processes, CPU scheduling, or potential hardware variations

MEMORY TEST (STRESS-NG) on Master:

- stress-ng –vm 1 –vm-bytes 1500M –timeout 60s –metrics-brief

```
stress-ng: info:  [19121] setting to a 1 min, 0 secs run per stressor
stress-ng: info:  [19121] dispatching hogs: 1 vm
stress-ng: info:  [19121] note: /proc/sys/kernel/sched_autogroup_enabled is 1 and this can impact scheduling throughput for processes not attached to a tty.
stress-ng: metrc: [19121] stressor       bogo ops real time  usr time  sys time   bogo ops/s     bogo ops/s
stress-ng: metrc: [19121]                           (secs)    (secs)    (secs)   (real time) (usr+sys time)
stress-ng: metrc: [19121] vm              510000     60.40     22.07     37.53      8444.08        8556.63
stress-ng: info:  [19121] skipped: 0
stress-ng: info:  [19121] passed: 1: vm (1)
stress-ng: info:  [19121] failed: 0
stress-ng: info:  [19121] metrics untrustworthy: 0
stress-ng: info:  [19121] successful run completed in 1 min, 0.40 secs
```

MEMORY TEST (STRESS-NG) on Node:

- stress-ng –vm 1 –vm-bytes 1500M –timeout 60s –metrics-brief

```
stress-ng: info:  [1503] setting to a 1 min, 0 secs run per stressor
stress-ng: info:  [1503] dispatching hogs: 1 vm
stress-ng: info:  [1503] note: /proc/sys/kernel/sched_autogroup_enabled is 1 and this can impact scheduling throughput for processes not attached to a tty.
stress-ng: metrc: [1503] stressor       bogo ops real time  usr time  sys time   bogo ops/s     bogo ops/s
stress-ng: metrc: [1503]                           (secs)    (secs)    (secs)   (real time) (usr+sys time)
stress-ng: metrc: [1503] vm              1787593     60.26     40.08     19.94     29663.83       29783.60
stress-ng: info:  [1503] skipped: 0
stress-ng: info:  [1503] passed: 1: vm (1)
stress-ng: info:  [1503] failed: 0
stress-ng: info:  [1503] metrics untrustworthy: 0
stress-ng: info:  [1503] successful run completed in 1 min, 0.27 secs
```

Comments on the results:

Metrics:

The memory stress test was performed using the stress-ng command with the vm parameter set to 1, meaning a single worker allocated and deallocated memory. The following metrics were measured:

- **bogo ops**: number of operations performed by the test.
- **real time (s)**: actual elapsed time to complete the test.
- **user time (s)**: CPU time spent in user mode.
- **sys time (s)**: CPU time spent in kernel mode.
- **bogo ops/s**: operations performed per second.

Comparison:

- The **Master** executed **510,000 bogo ops**, with a real-time duration of **37.53s**, user time of **22.07s**, and kernel time of **8444.08s**, achieving a rate of **8556.63 bogo ops/s**.
- The **Node** executed **1,787,539 bogo ops**, with a real-time duration of **19.94s**, user time of **49.80s**, and kernel time of **23,663.03s**, achieving a rate of **23,703.60 bogo ops/s**.

Conclusions:

The Node performed significantly more operations than the Master, with over three times the number of bogo ops and a much higher execution speed. This suggests that the Node has better memory performance

Even if the Master and Node are configured identically, differences in performance can arise due to several factors. The Master might be running additional background processes or services that consume CPU and memory resources, leading to lower efficiency. Another factor could be CPU scheduling and NUMA effects, where differences in CPU-to-memory access paths influence performance. The Node may have a more optimal memory allocation due to the way the kernel assigns resources based on workload conditions. Additionally, memory management and caching behavior can vary, even with identical setups. Differences in page allocation, swap usage, and caching efficiency may result in the Node utilizing memory more effectively, reducing latency and increasing throughput.

Disk I/O Test:

This section of the report focuses on the **Disk I/O performance test** on the local file system. Disk I/O (Input/Output) testing measures the system's ability to read and write data to storage devices, evaluating factors such as speed, efficiency, and overall performance. For this purpose, the IOzone package on Ubuntu will be used. The test will be conducted on both the Master and Node01 to assess and compare their disk performance.

The package has been retrieved from:

- sudo apt install iozone3 -y

IOZONE TEST MASTER:

- iozone -a -g 1G

```
	Iozone: Performance Test of File I/O
	        Version $Revision: 3.506 $
	        Compiled for 64 bit mode.
	        Build: linux-AMD64

	Contributors:William Norcott, Don Capps, Isom Crawford, Kirby Collins
	             Al Slater, Scott Rhine, Mike Wisner, Ken Goss
	             Steve Landherr, Brad Smith, Mark Kelly, Dr. Alain CYR,
	             Randy Dunlap, Mark Montague, Dan Million, Gavin Brebner,
	             Jean-Marc Zucconi, Jeff Blomberg, Benny Halevy, Dave Boone,
	             Erik Habbinga, Kris Strecker, Walter Wong, Joshua Root,
	             Fabrice Bacchella, Zhenghua Xue, Qin Li, Darren Sawyer,
	             Vangel Bojaxhi, Ben England, Vikentsi Lapa,
	             Alexey Skidanov, Sudhir Kumar.

	Run began: Fri Feb 14 14:11:52 2025

	Auto Mode
	Using maximum file size of 1048576 kilobytes.
	Command line used: iozone -a -g 1G
	Output is in kBytes/sec
	Time Resolution = 0.000001 seconds.
	Processor cache size set to 1024 kBytes.
	Processor cache line size set to 32 bytes.
	File stride size set to 17 * record size.
                                                            random    random      bkwd    record    stride
              kB  reclen    write  rewrite    read    reread    read     write      read   rewrite      read   fwrite frewrite   fread  freread
              64       4   866463  3363612 7293312  15972885 10402178   4274062   4274062   7100397   8182586  4564786   4274062 6421025 10821524
              64       8  2467108  5860307 15972885 10821524  9006179   4897948   5283570  10402178  15972885  7100397   7100397 12902017 20962191
              64      16  2067979  5389653 15972885 20962191 15972885   7100397   7940539   9318832  15972885  5283570   7100397  8182586 15972885
              64      32  2662899  4274062 30484297 34389688 20962191   5735102   6421025  10821524  12310336  5283570   7100397 10821524 15972885
              64      64  2561267  5860307 64000000 64000000 15972885   7940539   6421025   7100397  12902017  6421025   6421025 15972885 30484297
             128       4  2132084  5325799 14200794 14200794 10567140   6114306   6727225   9129573   9129573  5545860   6114306 11720614 12842051
             128       8  3785961  7082197 15881078 20804356 18012359   4012317   9795896   9795896  14200794  7082197   7082197 14200794 18012359
             128      16  3536566  7476717 24620673 24620673 20804356   8548124   8548124  12842051  18637664  7917784   8036304 15881078 21643049
             128      32  3759450  8548124 31945770 31945770 25804035   9287508  11720614  11720614  18637664  7082197   5545860 15881078 18012359
             128      64  3759450  8414153 41924383 45475583 25804035   9977956  11720614   9795896  16365173  4557257   9795896 18012359 25804035
```

The screenshot shows the results of an **IOzone** test. The command above runs an automatic mode test (-a) with a maximum file size of **1GB** (-g 1G). The results table displays various read and write performance metrics in **(KB/s)** for different file sizes and record sizes.

Key parameters include **sequential and random read/write speeds, re-read/re-write performance, and various access patterns (stride, backward, record, etc.)**. The numbers in the table indicate throughput for each operation, where higher values generally mean better performance.

From the results, **larger record sizes tend to yield higher throughput**, which is expected as larger writes reduce the overhead of multiple small operations. These results help in evaluating disk performance, optimizing storage configurations, and identifying bottlenecks in file system operations.

IOZONE TEST NODE:

- iozone -a -g 1G

```
	Iozone: Performance Test of File I/O
	        Version $Revision: 3.506 $
	        Compiled for 64 bit mode.
	        Build: linux-AMD64

	Contributors:William Norcott, Don Capps, Isom Crawford, Kirby Collins
	             Al Slater, Scott Rhine, Mike Wisner, Ken Goss
	             Steve Landherr, Brad Smith, Mark Kelly, Dr. Alain CYR,
	             Randy Dunlap, Mark Montague, Dan Million, Gavin Brebner,
	             Jean-Marc Zucconi, Jeff Blomberg, Benny Halevy, Dave Boone,
	             Erik Habbinga, Kris Strecker, Walter Wong, Joshua Root,
	             Fabrice Bacchella, Zhenghua Xue, Qin Li, Darren Sawyer,
	             Vangel Bojaxhi, Ben England, Vikentsi Lapa,
	             Alexey Skidanov, Sudhir Kumar.

	Run began: Sat Feb 15 16:21:38 2025

	Auto Mode
	Using maximum file size of 1048576 kilobytes.
	Command line used: iozone -a -g 1G
	Output is in kBytes/sec
	Time Resolution = 0.000001 seconds.
	Processor cache size set to 1024 kBytes.
	Processor cache line size set to 32 bytes.
	File stride size set to 17 * record size.
                                                            random    random      bkwd    record    stride
              kB  reclen    write  rewrite    read    reread    read     write      read   rewrite      read   fwrite frewrite   fread  freread
              64       4   496362  4564786 15972885  12310336  9006179   4564786   5860307   4897948   7100397   863676   4564786  7940539  2772930
              64       8  2662899  5860307 34389688  22737791 12902017   4988978   6421025   7940539   9006179  6421025   780776  10821524 15972885
              64      16  2923952  5735102 30484297  30484297 15972885    969761   7100397   7940539  10821524  5389653   929467  10402178 20962191
              64      32  2772930  4988978 55857768  30484297 20962191   4897948   7940539    983980  12902017  4897948  7100397  10402178 15972885
              64      64  2561267  5283570 64000000  64000000 20962191   7100397   7100397   9318832  10821524  7100397  7100397  10402178 22737791
             128       4  2969325  6045455 12842051  14200794  9977956   6727225   6406138   9129573   7082197  5074121  5545860   9977956 11720614
             128       8  2714132  6727225 21643049  25804035 18012359   8036304   8548124  11470204  14200794  7176872  6727225   9129573 18012359
             128      16  3785961  7476717  2098744   2098744 21643049   9129573  10779307  14200794  18637664  4934216  7582312  14200794 21643049
             128      32  4012317  7082197 31945770  25804035 25804035   9795896   1183041  12842051  18637664  9129573  9129573  16365173  2132084
             128      64  3982553  9129573 41924383  41924383 25804035   9795896  12842051  12842051  15881078  6406138 10567140  15881078 31945770
```

The IOzone test results shown in the screenshot were generated using the command iozone -a -g 1G, which runs in automatic mode while testing various file and record sizes with a maximum file size of 1GB.

The results provide throughput values in KB/s for different read and write operations, including *sequential access*, *random access*, *backward reads*, *and strided reads*. As expected, performance improves with larger record sizes since fewer operations are required per file. Sequential read and write speeds are significantly higher than random access speeds, highlighting the inefficiencies of non-contiguous disk access. The results also show that record rewrite performance is relatively strong, indicating efficient handling of repeated write operations. However, backward reads and strided accesses show lower throughput, suggesting that these access patterns may not be optimized for peak performance.

For a **Node-based** system, these results are important in assessing file system efficiency, optimizing storage configurations, and understanding caching behavior in **high I/O workloads**. In scenarios involving databases or large-scale applications, evaluating these performance metrics can help identify potential bottlenecks.

Network Test:

After completing the I/O performance tests, **connectivity tests** between nodes and the master was conducted to evaluate network performance. For this purpose, tools such as *HyPerf3*, *Netcat*, and Ping was utilized to assess packet reception over TCP/IP, measure throughput, and record latency. The primary objective of this analysis was to ensure the accurate transmission and reception of data while obtaining key network performance metrics. The results are presented in detail, covering communication from Node to Master, Master to Node, and between the two nodes within our *ClusterVimNet* environment.

Ping Tests:

In this section, we conduct a **ping test** to evaluate network connectivity and latency between nodes within our environment. The test is performed across three distinct communication paths: from the master node to a worker node, from a worker node to the master, and between Node01 and Node02. Each test involves sending ICMP packets, each containing 64 bytes of data, to measure response times and packet reception. The transmitted packets are sequentially numbered within an **ICMP** sequence, and the results are analyzed based on minimum, average, and maximum round-trip times (**RTT**). Additionally, we assess the total transmission time and the percentage of successfully received packets. Screenshots of each test scenario are provided to verify the actual reception of packets. The following command has been executed:

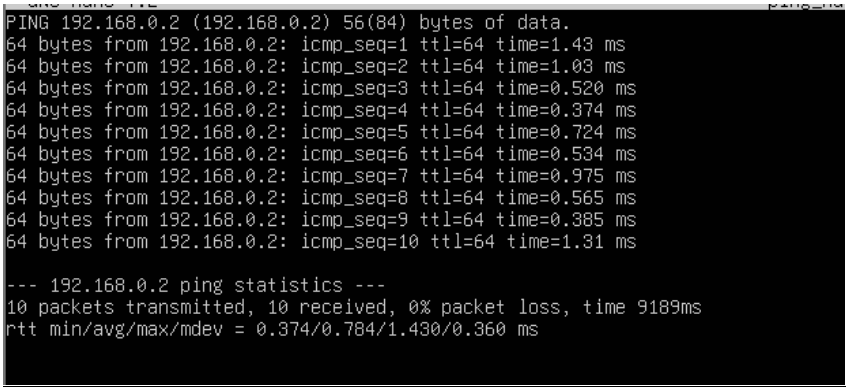
- ping -c 10 192.168.0.x

ping -> The command-line utility used to test network connectivity between a source and a destination by sending ICMP echo request packets.

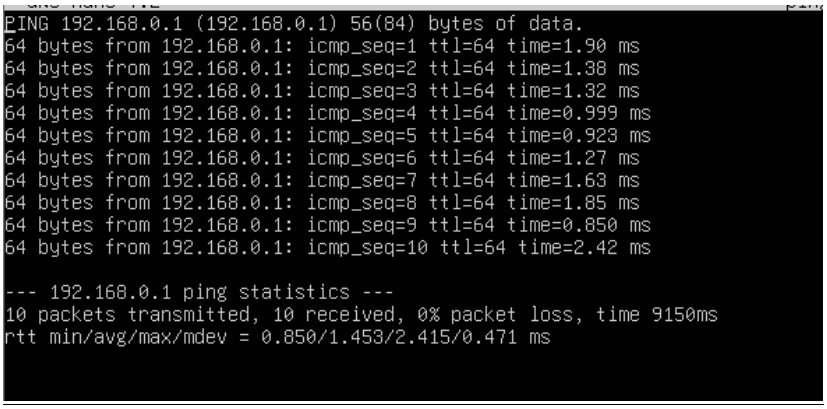
-c 10 -> This option specifies the number of packets to send. In this case, 10 packets will be transmitted before the command terminates.

192.168.0.x → This is the target IP address being tested. The placeholder x represents the specific last octet of the IP address for the destination node.

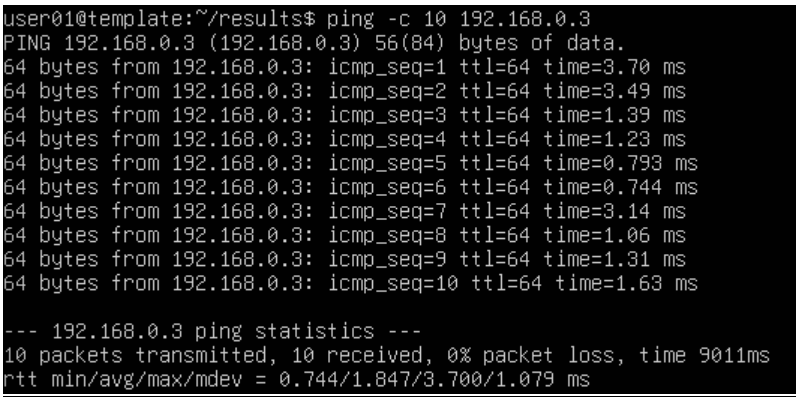
Master -> Node



Node -> Master



Node01 -> Node02



Iperf3 Network Bandwidth Test:

We will now utilize **HyPerf3** to evaluate network throughput between the nodes within the *clustervimnet* environment. This process requires the installation of the necessary dependencies, including the **HyPerf3** package, on an Ubuntu system. Once the tests are conducted, the results will be analyzed based on transmission time and bandwidth measurements, allowing for a detailed assessment of network performance.

Installing packages:

- sudo apt update && sudo apt install -y iperf3

Master -> Node

```
Connecting to host 192.168.0.2, port 5201
[  5] local 192.168.0.1 port 35280 connected to 192.168.0.2 port 5201
[ ID] Interval           Transfer     Bitrate         Retr  Cwnd
[  5]   0.00-1.00   sec  83.6 MBytes   701 Mbits/sec   90    334 KBytes
[  5]   1.00-2.00   sec  53.8 MBytes   451 Mbits/sec   56    273 KBytes
[  5]   2.00-3.00   sec  35.6 MBytes   299 Mbits/sec    0    359 KBytes
[  5]   3.00-4.00   sec  37.4 MBytes   314 Mbits/sec   45    304 KBytes
[  5]   4.00-5.00   sec  36.4 MBytes   305 Mbits/sec    0    385 KBytes
[  5]   5.00-6.00   sec  34.9 MBytes   293 Mbits/sec    0    451 KBytes
[  5]   6.00-7.00   sec  36.8 MBytes   308 Mbits/sec  135    269 KBytes
[  5]   7.00-8.00   sec  66.5 MBytes   558 Mbits/sec   45    305 KBytes
[  5]   8.00-9.00   sec  51.6 MBytes   433 Mbits/sec   45    317 KBytes
[  5]   9.00-10.00  sec  33.4 MBytes   280 Mbits/sec    0    389 KBytes
- - - - - - - - - - - - - - - - - - - - - - - - -
[ ID] Interval           Transfer     Bitrate         Retr
[  5]   0.00-10.00  sec   470 MBytes   394 Mbits/sec  416             sender
[  5]   0.00-10.03  sec   468 MBytes   391 Mbits/sec                  receiver

iperf Done.
```

The throughput test was conducted using **iPerf3** to evaluate network performance between the nodes. Initially, the receiving node was set to listening mode by running the command

- iperf3 -s

allowing it to accept incoming connections. The master node then initiated the test by transmitting data to the receiving node using

- iperf3 -c 192.168.0.2.

The results indicate a fluctuating bandwidth throughout the test, with variations in data transfer rates across different time intervals. The highest observed throughput occurred in the first second, reaching **701 Mbits/sec**, while subsequent measurements demonstrated variations, with some intervals dropping as low as **200 Mbits/sec**. The overall average bitrate over the 10-second duration was **394 Mbits/sec** on the sender side and **391 Mbits/sec** on the receiver side. The test also recorded **416 retransmissions**, which may suggest network congestion or suboptimal performance conditions.

Node -> Master

```
Connecting to host 192.168.0.1, port 5201
[  5] local 192.168.0.2 port 49722 connected to 192.168.0.1 port 5201
[ ID] Interval           Transfer     Bitrate         Retr  Cwnd
[  5]   0.00-1.08   sec  57.8 MBytes   448 Mbits/sec  180    300 KBytes
[  5]   1.08-2.08   sec  67.5 MBytes   567 Mbits/sec  180    520 KBytes
[  5]   2.08-3.07   sec  44.6 MBytes   378 Mbits/sec    0    520 KBytes
[  5]   3.07-4.06   sec  45.8 MBytes   389 Mbits/sec    0    522 KBytes
[  5]   4.06-32.83  sec  1.36 GBytes   406 Mbits/sec  1803    218 KBytes
- - - - - - - - - - - - - - - - - - - - - - - - -
[ ID] Interval           Transfer     Bitrate         Retr
[  5]   0.00-32.83  sec  1.57 GBytes   411 Mbits/sec  2163             sender
[  5]   0.00-32.83  sec  1.56 GBytes   409 Mbits/sec                  receiver

iperf Done.
```

Then, establishing a connection from **Node** to the **Master** node. The receiving node was initialized in listening mode using iperf3 -s, while Node acted as the client, transmitting data via iperf3 -c 192.168.0.1.

The results indicate an initial peak bitrate of **567 Mbits/sec**, followed by fluctuations throughout the test duration. The average throughput recorded over the entire session was **411 Mbits/sec** on the sender side and **409 Mbits/sec** on the receiver side. Despite relatively stable bandwidth performance, the transmission recorded **2,163 retransmissions**, suggesting potential congestion or packet loss affecting data integrity.

Node01 -> Node02

```
Connecting to host 192.168.0.3, port 5201
[  5] local 192.168.0.2 port 36356 connected to 192.168.0.3 port 5201
[ ID] Interval           Transfer     Bitrate         Retr  Cwnd
[  5]   0.00-1.00   sec  33.6 MBytes   282 Mbits/sec   90    512 KBytes
[  5]   1.00-2.00   sec  28.5 MBytes   239 Mbits/sec    0    574 KBytes
[  5]   2.00-3.00   sec  35.2 MBytes   296 Mbits/sec   45    461 KBytes
[  5]   3.00-4.00   sec  36.9 MBytes   309 Mbits/sec   45    392 KBytes
[  5]   4.00-5.00   sec  38.5 MBytes   323 Mbits/sec   45    325 KBytes
[  5]   5.00-6.00   sec  40.2 MBytes   338 Mbits/sec    0    409 KBytes
[  5]   6.00-7.00   sec  28.2 MBytes   237 Mbits/sec    0    461 KBytes
[  5]   7.00-8.00   sec  37.4 MBytes   314 Mbits/sec   87    361 KBytes
[  5]   8.00-9.00   sec  61.2 MBytes   514 Mbits/sec    0    423 KBytes
[  5]   9.00-10.00  sec  66.8 MBytes   560 Mbits/sec  144    313 KBytes
- - - - - - - - - - - - - - - - - - - - - - - - -
[ ID] Interval           Transfer     Bitrate         Retr
[  5]   0.00-10.00  sec   407 MBytes   341 Mbits/sec  456             sender
[  5]   0.00-10.01  sec   402 MBytes   337 Mbits/sec                  receiver

iperf Done.
```

A throughput test was conducted using **iPerf3**, establishing a connection from **Node01** to **Node02**. The receiving node was set to listening mode with iperf3 -s, while Node02 initiated the test as the client using iperf3 -c 192.168.0.3.

The results show an initial bitrate of **282 Mbits/sec**, followed by variations across different time intervals, with throughput reaching a peak of **560 Mbits/sec** in the final second. The average transmission rate over the 10-second test was **341 Mbits/sec** on the sender side and **337 Mbits/sec** on the receiver side. A total of **456 retransmissions** were recorded, suggesting some degree of network congestion or packet loss during the test.

These performance metrics provide insight into the network stability and data transfer efficiency between Node01 and Node02, indicating relatively consistent throughput with minor fluctuations.

Netcat throughput test:

Finally, network throughput was evaluated using **Netcat** as the testing tool. Consistent with the previous assessments, three key test scenarios were examined, and the results were analyzed to assess data transmission performance and network efficiency.

To perform this test, the following command has been executed: (**sender**)

- time dd if=/dev/zero/ bs=1M count = 100 | nc 192.168.0.x 5000

receiver, should listen to the communication using:

- nc -l -p 5000 > /dev/null

Breakdown of Command Parameters:

- **time** → Measures the execution time of the command, providing insights into transmission duration.
- **dd if=/dev/zero** → Uses the dd command to generate a stream of data from /dev/zero, which produces a continuous flow of null bytes.
- **bs=1M** → Sets the block size to **1 megabyte**, meaning data is processed in 1MB chunks.
- **count=100** → Specifies that **100 blocks** (each of 1MB) will be transmitted, resulting in a total data transfer of **100MB**.
- **| (pipe)** → Directs the output of dd to the next command in the pipeline.
- **nc 192.168.0.x 5000** → Uses **Netcat (nc)** to send the data to the specified IP address (192.168.0.x) on port **5000**, where a receiving Netcat instance is expected to be listening.

This command measures the time required to transfer **100MB** of data between nodes, allowing for an estimation of network throughput.

Master -> Node

```
100+0 records in
100+0 records out
104857600 bytes (105 MB, 100 MiB) copied, 3.56112 s, 29.4 MB/s
^C

real    0m5.635s
user    0m0.013s
sys     0m0.124s
```

Node -> Master

```
100+0 records in
100+0 records out
104857600 bytes (105 MB, 100 MiB) copied, 2.39807 s, 43.7 MB/s
^C

real    0m8.709s
user    0m0.001s
sys     0m0.140s
```

Node01 -> Node02

```
100+0 records in
100+0 records out
104857600 bytes (105 MB, 100 MiB) copied, 2.95795 s, 35.4 MB/s
```

Since all three components are configured identically, the **variations in transfer speeds** likely stem from factors beyond basic configuration differences. The asymmetry between Master → Node (29.4 MB/s) and Node → Master (43.7 MB/s) suggests potential discrepancies in network routing, background load, or hardware performance differences. The Node01 → Node02 transfer (35.4 MB/s) falls between the two, indicating that link-specific conditions, transient congestion, or even kernel-level optimizations might be influencing throughput despite uniform configurations.

Part II: Container Performance Test

This section of the report examines the performance tests conducted on a set of containers. As with the virtual machines, the containers are initialized through a predefined configuration file, ensuring a consistent setup. An internal network is established to facilitate communication between them, while their hardware resources are deliberately constrained to assess performance under limited conditions.

Setup:

The initial phase involves detailing the container initialization process and specifying the required software components. **Docker Desktop** is utilized as the primary platform to enable an efficient containerization environment. (https://www.docker.com/products/docker-desktop/)

To define and configure a set of containers, both a *docker-compose.yaml* file and a *Dockerfile* are employed. The *docker-compose.yaml* file **orchestrates** the creation of three distinct containers—*hpcc_master*, *hpcc_node01*, and *hpcc_node02*—designed for performance testing. These containers are interconnected through an internal network, designated as *my_custom_network*, which operates within the **192.168.1.0/24 subnet**. Each container is assigned a unique alias (*Node01*, *Node02*, and *Master*) and a corresponding IP address within this subnet.

In alignment with the predefined constraints, each container is allocated limited hardware resources, specifically 2 GB of memory and 2 CPU cores. The containers are instantiated using **Ubuntu 24.04**, mirroring the virtual machine configurations.

Docker-compose.yaml

```yaml
version: '3'
▷Run All Services
services:
  ▷Run Service
  master:
    build: .
    hostname: master
    container_name: hpcc_master
    volumes:
      - ./shared:/shared
    ports:
      - "2222:22"
    deploy:
      resources:
        limits:
          cpus: '2'
          memory: 2G
    networks:
      my_custom_network:
        ipv4_address: 192.168.1.2

  ▷Run Service
  node01:
    build: .
    hostname: node01
    container_name: hpcc_node01
    volumes:
      - ./shared:/shared
    deploy:
      resources:
        limits:
          cpus: '2'
          memory: 2G
    networks:
      my_custom_network:
        ipv4_address: 192.168.1.3

  ▷Run Service
  node02:
    build: .
    hostname: node02
    container_name: hpcc_node02
    volumes:
      - ./shared:/shared
    deploy:
      resources:
        limits:
          cpus: '2'
          memory: 2G
    networks:
      my_custom_network:
        ipv4_address: 192.168.1.4

networks:
  my_custom_network:
    driver: bridge
    ipam:
      config:
        - subnet: 192.168.1.0/24
```

The **Dockerfile** serves as a blueprint for the automated provisioning of essential **libraries** and tools required for performance testing within the containers. Its primary function is to define a structured sequence of **commands** that are executed during the container build process, ensuring the installation and configuration of the necessary dependencies.

In this context, the **Dockerfile** facilitates the pre-installation of benchmarking utilities such as *Netcat*, *IOzone*, and *Sysbench*, which are essential for assessing network performance, disk I/O, and system resource utilization, respectively. By specifying the installation of these tools within the container build process, the **Dockerfile** guarantees a consistent and reproducible environment for performance evaluations. Furthermore, it ensures that all dependencies are correctly set up before container deployment, eliminating the need for manual configuration post-deployment and enhancing the reliability of the testing framework.

Dockerfile

```
FROM ubuntu:24.04

RUN apt-get update && apt-get install -y \
    hpcc \
    openmpi-bin \
    openmpi-common \
    stress-ng \
    sysbench \
    iozone3 \
    iperf3 \
    openssh-server \
    iputils-ping \
    && rm -rf /var/lib/apt/lists/*

RUN mkdir -p /var/run/sshd
RUN mkdir -p /root/.ssh
RUN chmod 700 /root/.ssh

RUN ssh-keygen -t rsa -N '' -f /root/.ssh/id_rsa
RUN cp /root/.ssh/id_rsa.pub /root/.ssh/authorized_keys

RUN echo "StrictHostKeyChecking no" >> /etc/ssh/ssh_config
RUN touch /root/.ssh/known_hosts

RUN mkdir -p /shared

RUN mkdir -p /etc/hpcc
COPY HPL.dat /etc/hpcc/
RUN echo "node01 slots=1\nnode02 slots=1" > /etc/hpcc/hostfile

EXPOSE 22

ENV OMPI_ALLOW_RUN_AS_ROOT=1
ENV OMPI_ALLOW_RUN_AS_ROOT_CONFIRM=1

CMD ["/usr/sbin/sshd", "-D"]
```

To enable **SSH** access, the configuration creates the necessary directories, generates an SSH key pair, and sets up **passwordless** authentication by copying the public key to *authorized_keys*. It also modifies the **SSH** client configuration to disable strict host key checking and preemptively creates an empty *known_hosts* file to prevent potential authentication prompts.

A shared directory is created at /shared to facilitate data exchange between containers. The */etc/hpcc* directory is also created, and the **HPL.dat** file is copied into it. Additionally, a hostfile defining the compute nodes, node01 and node02, each assigned one slot, is generated to support **MPI-based** execution.

The container exposes port **22** to allow SSH connections. The environment variables **OMPI_ALLOW_RUN_AS_ROOT** and **OMPI_ALLOW_RUN_AS_ROOT_CONFIRM** are set to permit running MPI processes as the root user. Finally, the container launches the **OpenSSH** daemon (sshd) in the foreground as its primary process.

Usage

Build Docker Image

The workflow begins with the creation of a **Docker** image using the docker build command. This command constructs an image based on the instructions specified in the Dockerfile, which includes the base operating system, application dependencies, and necessary configurations. The *-t* flag is used to tag the image with a recognizable name, such as ubuntu-performance, facilitating easier reference in subsequent commands.

- docker-build -t hpcc_image .

Run containers

Once the image is successfully built, the next step involves deploying the containerized environment using docker-compose up -d. Docker Compose allows for the orchestration of multi-container applications by defining services, networks, and volumes within a *docker-compose.yml* file. The -d flag ensures that the containers run in detached mode, enabling the user to continue other operations while the services remain active in the background. This approach streamlines the management of complex environments by automating the startup sequence and dependency resolution.

- docker-compose up -d

Check running containers

After deployment, verifying the status of running containers is crucial. The docker ps command lists all active containers, displaying essential details such as container *IDs*, assigned *ports*, and running *statuses*. This information aids in monitoring the deployment and diagnosing potential issues related to network configurations or application behavior.

- docker ps

Interact with a running container

To interact with a running **container**, the docker exec -it hpcc_master bash command is employed. This command spawns an interactive terminal session within the specified container, allowing direct **access** to its filesystem and execution of administrative commands. The *-it* flags ensure that the session remains interactive, providing real-time input and output, which is particularly useful for debugging and manual configuration adjustments.

- docker exec -it hpcc_master bash

Terminate and remove container

Once the application lifecycle is complete or the environment requires a reset, the docker-compose down command is used to stop and remove the containers, networks, and volumes defined in the

docker-compose.yml file. This command ensures that no residual processes or dependencies remain active, preventing resource leakage and maintaining a clean system state.

- docker-compose down

Remove a defined network

Finally, the docker network rm my_custom_network command is executed to **remove** a custom-**defined network**, if applicable. Docker automatically creates networks to facilitate inter-container communication, but in scenarios where a specific network was manually defined and is no longer required, this command helps in reclaiming system resources and avoiding **network conflicts** in future deployments.

- docker network rm my_custom_network

This sequence of commands exemplifies a structured approach to container management, from image creation to cleanup, ensuring a controlled and efficient environment for deploying containerized applications.

CPU Test HPC Challenge Benchmark:

Once the containerized system has been properly set up, performance testing must be conducted to ensure that the system is functioning correctly and to collect critical performance metrics. One of the key tools used for this evaluation is the **HPC Challenge Benchmark (HPCC**), a comprehensive benchmarking suite designed to assess the computational capabilities of high-performance computing (HPC) systems. HPCC operates at the system level, providing insights into *computational performance*, *memory bandwidth*, and *network communication efficiency*, similar to the benchmarking procedures previously conducted for **virtual machines** (VMs).

To facilitate this testing process, the HPCC benchmark suite has been integrated into the containerized environment through the **Dockerfile**. The necessary dependencies, including required libraries and configuration files, have been pre-installed during the container build process. This ensures that once the container is deployed, the benchmarking tests can be executed without additional setup, allowing for a streamlined evaluation of system performance.

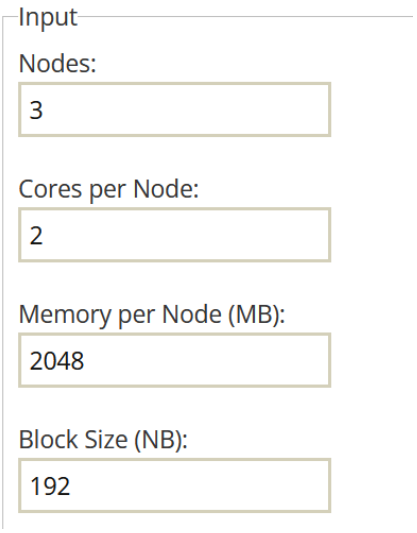
By running HPCC within the containerized infrastructure, we can analyze key performance indicators such as floating-point operations per second (**FLOPS**), **memory throughput**, and inter-process communication efficiency. These metrics provide valuable insights into the computational efficiency and scalability of the containerized system, enabling comparisons with traditional **VM-based** setups and identifying potential optimizations for improved performance.

HPL.dat

The **HPL.dat** file was configured according to the guidelines provided in "Tweak HPL Parameters". The configuration parameters include:

- **Number of nodes**: 3
- **Cores per node**: 2
- **Total memory per node**: 2048 MB
- **Block size**: Specified based on tuning recommendations (192)

These settings were used as input to optimize **High-Performance Linpack (HPL)** execution.



Run test

- mpirun --allow-run-as-root -np 2 --display-map --map-by node \
  --hostfile /etc/hpcc/hostfile \
  --mca btl tcp,self \
  --mca btl_tcp_if_include eth0 \
  --wdir /etc/hpcc \
  --output-filename /etc/hpcc/hpcc_summary.txt \
   /usr/bin/hpcc

Once inside the **master** container, the specified **MPI-based HPCC benchmark** test was executed to evaluate computational performance across multiple nodes. The command utilizes *mpirun* to distribute workloads across two processes, mapping them by node to ensure execution on separate hosts. The *--hostfile /etc/hpcc/hostfile* option defines the participating compute nodes, while *--mca btl tcp,self and --mca btl_tcp_if_include* eth0 configure *OpenMPI* to use *TCP-based* communication over the specified network interface. The working directory is set *to /etc/hpcc*, ensuring that input files such as HPL.dat and configuration settings are correctly referenced. The execution results are redirected to *hpcc_summary.txt*, preserving benchmark outputs for analysis. This test measures floating-point performance, memory bandwidth, and communication latency, providing insights into cluster efficiency under high-performance workloads.

Results:

The HPCC benchmark results provide key performance metrics that assess computational efficiency, memory bandwidth, and communication latency within the containerized environment. High **GFLOPS** values in the **HPL** benchmark indicate effective floating-point computation, while **STREAM** benchmarks evaluate memory throughput, reflecting data movement efficiency. Low latency and high bandwidth in **PTRANS** and **MPI** tests highlight optimized inter-container communication, a crucial factor for distributed workloads. The results underscore how **containerization** introduces **minimal overhead**, ensuring near-native performance by leveraging resource isolation and efficient networking, making it a viable solution for scalable, high-performance computing workloads.

General System Test:

After completing the HPL benchmark, a general system performance test was conducted using the *stress-ng* and *sysbench* utilities. These tests were executed on the *Container1* to assess system stability and resource utilization. The results, along with the command-line execution and parameter configurations, will be analyzed in the following sections. First, we need to execute the container to perform the test.

- docker exec -it hpcc_master

CPU TEST (SYSBENCH):

Sysbench evaluates **CPU** performance by calculating prime numbers up to 20,000 using two parallel threads. The key metrics include an event processing rate of 3,089.76 events per second and a total execution time of approximately 10 seconds. The latency distribution shows minimal variation, with an average latency of 0.65 ms and a maximum of 1.51 ms. These results indicate that the system maintains consistent computational efficiency under multithreaded workloads, demonstrating stable CPU performance with negligible thread fairness deviation.

- sysbench cpu --cpu-max-prime=20000 --threads=2 run

```
Initializing worker threads...

Threads started!

CPU speed:
    events per second:  3089.76

General statistics:
    total time:                          10.0002s
    total number of events:              30901

Latency (ms):
         min:                                    0.61
         avg:                                    0.65
         max:                                    1.51
         95th percentile:                        0.68
         sum:                                19991.76

Threads fairness:
    events (avg/stddev):           15450.5000/3.50
    execution time (avg/stddev):   9.9959/0.00
```

MEMORY TEST (SYSBENCH):

The second test measures **memory** performance using **Sysbench** by allocating and processing a total of 2GB of memory. The system achieves a transfer rate of 2931.53 MiB/sec, highlighting efficient memory bandwidth utilization. The latency is exceptionally low, with an average of 0.221 microseconds, indicating near-instantaneous memory access. The lack of significant latency spikes confirms stable memory performance, ensuring that memory-bound operations are handled with minimal overhead, crucial for high-performance computing applications.

- sysbench memory –memory-total-size=2G run

```
Threads started!

Total operations: 2097152 (9453090.11 per second)

2048.00 MiB transferred (9231.53 MiB/sec)


General statistics:
    total time:                          0.2210s
    total number of events:              2097152

Latency (ms):
         min:                                    0.00
         avg:                                    0.00
         max:                                    0.06
         95th percentile:                        0.00
         sum:                                   87.11

Threads fairness:
    events (avg/stddev):           2097152.0000/0.00
    execution time (avg/stddev):   0.0871/0.00
```

CPU TEST (STRESS-NG):

For **CPU** stress testing, **Stress-NG employs** the matrix product method with two CPU stressors for a duration of 60 seconds. The system processes approximately 71,851 bogo operations per second, with a total real-time execution of 60 seconds and accumulated user+system time of 1,197.58 seconds. These results demonstrate the system's ability to sustain heavy CPU workloads, effectively utilizing computational resources while maintaining predictable execution times. The relatively high system time compared to user time suggests an intensive computational burden, testing processor efficiency under synthetic loads.

- stress-ng –cpu 2 –cpu-method matrixprod –timeout 60s –metrics-brief

```
stress-ng: info:  [26] setting to a 60 second run per stressor
stress-ng: info:  [26] dispatching hogs: 2 cpu
stress-ng: info:  [26] successful run completed in 60.00s
stress-ng: info:  [26] stressor       bogo ops real time  usr time  sys time   bogo ops/s     bogo ops/s
stress-ng: info:  [26]                           (secs)    (secs)    (secs)   (real time) (usr+sys time)
stress-ng: info:  [26] cpu                71851     60.00    119.94      0.00      1197.58         599.06
```

MEMORY TEST (STRESS-NG):

The final test evaluates **memory** stress using **Stress-NG** with a single virtual memory stressor, allocating and manipulating 1.5GB of memory over 60 seconds. The system achieves 6,799,569 bogo operations with a user+system time accumulation of 113,499.49 seconds. The low system time relative to user time suggests efficient memory access patterns, and the high bogo operation count reflects sustained memory throughput. These results indicate that the system effectively manages large-scale memory operations under stress, maintaining stable memory performance without significant slowdowns.

- stress-ng –vm 1 –vm-bytes 1500M –timeout 60s –metrics-brief

```
root@6ee8bb165df9:/# stress-ng --vm 1 --vm-bytes 1500M --timeout 60s --metrics-brief
stress-ng: info:  [29] setting to a 60 second run per stressor
stress-ng: info:  [29] dispatching hogs: 1 vm
stress-ng: info:  [29] successful run completed in 60.03s (1 min, 0.03 secs)
stress-ng: info:  [29] stressor       bogo ops real time  usr time  sys time   bogo ops/s     bogo ops/s
stress-ng: info:  [29]                           (secs)    (secs)    (secs)   (real time) (usr+sys time)
stress-ng: info:  [29] vm              6797569     60.03     48.06     11.91    113229.77      113349.49
```

Disk I/O Test:

This section of the report focuses on the **Disk I/O performance test** on the local file system. Disk I/O (Input/Output) testing measures the system's ability to read and write data to storage devices, evaluating factors such as speed, efficiency, and overall performance. For this purpose, the IOzone package on Ubuntu will be used. The test will be conducted on Container1 to assess disk performance.

- docker exec -it container1

```
        Iozone: Performance Test of File I/O
                Version $Revision: 3.489 $
                Compiled for 64 bit mode.
                Build: linux-AMD64

        Contributors:William Norcott, Don Capps, Isom Crawford, Kirby Collins
                     Al Slater, Scott Rhine, Mike Wisner, Ken Goss
                     Steve Landherr, Brad Smith, Mark Kelly, Dr. Alain CYR,
                     Randy Dunlap, Mark Montague, Dan Million, Gavin Brebner,
                     Jean-Marc Zucconi, Jeff Blomberg, Benny Halevy, Dave Boone,
                     Erik Habbinga, Kris Strecker, Walter Wong, Joshua Root,
                     Fabrice Bacchella, Zhenghua Xue, Qin Li, Darren Sawyer,
                     Vangel Bojaxhi, Ben England, Vikentsi Lapa,
                     Alexey Skidanov, Sudhir Kumar.

        Run began: Sun Feb 16 15:56:11 2025

        Auto Mode
        Using maximum file size of 1048576 kilobytes.
        Command line used: iozone -a -g 1G
        Output is in kBytes/sec
        Time Resolution = 0.000001 seconds.
        Processor cache size set to 1024 kBytes.
        Processor cache line size set to 32 bytes.
        File stride size set to 17 * record size.
                                                            random    random     bkwd    record    stride
              kB  reclen    write  rewrite    read    reread    read     write     read   rewrite      read   fwrite frewrite   fread  freread
              64       4   821391  2561267  2561267 10821524  7940539  2203800  1828508   3541098   5389653  3363612  3588436  6421025  9318832
              64       8   998622  3738358  7100397 12902017  5283570  3738358  4564786   7940539  15972885  4897948  4564786  6421025  4897948
              64      16  1648808  4274062  7100397  7940539 10402178  6421025  4897948   9318832   6421025  2923952  5389653  7100397 10402178
              64      32  1679761  5283570 10821524 15972885 10402178  7100397  6271021   5389653  10821524  3203069  7940539 10821524 12902017
              64      64   801764  6421025 15972885 64000000 55857768  5389653  6421025   7100397   8182586  6421025  3791156  5860307  6271021
             128       4  1471662  5325799  8548124 10567140  7917784  4267461  4557257   9795896   7082197  3536566  4759253  6406138 10567140
             128       8  2464907  5325799  7917784  9795896 14200794  7476717  3560017   9795896  18637664  7176872  5122535  4135958 12842051
             128      16  2409592  6406138  7082197  9129573 24620673  7582312  7082197  14200794  21643049  7082197  5847904  6406138 12542043
             128      32  2784517  5784891  7476717 11470204 24620673  5122535  7176872  10779307  14200794  3759450  7176872  9129573 12842051
             128      64  2558895  6727225 12542043 18012359 18012359  8548124  6727225   9129573   9129573  5122535  8414153 10567140 12842051
```

The **Iozone** test executed with the -a -g 1G parameters performs a comprehensive file I/O benchmark within the **Docker** container, analyzing read, write, and access patterns across different file sizes up to 1GB. The test operates in *auto* mode, systematically measuring sequential and random access latencies, rewrite efficiency, and various read/write strategies. The dataset surpasses typical processor cache sizes, ensuring that the results reflect genuine disk or storage performance rather than *CPU* caching effects.

The results show that sequential read and write operations achieve high throughput, with read speeds exceeding **10,821,524 kB/s** for larger block sizes and sequential writes maintaining substantial performance. Random read and write operations **exhibit lower bandwidth**, as expected, given the increased seek times and fragmentation overhead. The random read performance stabilizes around 7,904,539 kB/s, while random write operations register slightly lower speeds, influenced by storage medium characteristics and filesystem journaling overhead. **Strided** access patterns indicate performance consistency, with stride read speeds averaging 5,389,653 kB/s, suggesting efficient large-block handling.

These findings provide insights into the containerized environment's I/O capabilities, revealing that the underlying storage subsystem performs efficiently under controlled workloads. The performance profile suggests that the container's filesystem configuration and storage backend introduce minimal bottlenecks, supporting demanding data-intensive applications.

Network Test:

After completing the I/O performance tests, **connectivity tests** between containers was conducted to evaluate network performance. For this purpose, I utilized tools such as *HyPerf3*, *Netcat*, and Ping to assess packet reception over TCP/IP, measure throughput, and record latency. The primary objective of this analysis was to ensure the accurate transmission and reception of data while obtaining key network performance metrics. The results are presented in detail, covering communication between the two containers within our my_*custom_network* environment.

Ping Test

The **ping test** evaluates basic network connectivity and latency between containers. The command ping -c 10 192.168.1.3 sends ten ICMP echo requests from the hpcc_master (192.168.1.2) to the target container (192.168.1.3). The results indicate stable connectivity, with minimal packet loss and consistent round-trip times. The reported latency values are crucial for assessing response delays, which in this test appear to be low, suggesting an efficient internal network configuration without significant congestion or interference.

- ping -c 10 192.168.1.3

```
root@6ee8bb165df9:/# ping -c 10 192.168.1.3
PING 192.168.1.3 (192.168.1.3) 56(84) bytes of data.
64 bytes from 192.168.1.3: icmp_seq=1 ttl=64 time=0.114 ms
64 bytes from 192.168.1.3: icmp_seq=2 ttl=64 time=0.083 ms
64 bytes from 192.168.1.3: icmp_seq=3 ttl=64 time=0.053 ms
64 bytes from 192.168.1.3: icmp_seq=4 ttl=64 time=0.043 ms
64 bytes from 192.168.1.3: icmp_seq=5 ttl=64 time=0.051 ms
64 bytes from 192.168.1.3: icmp_seq=6 ttl=64 time=0.051 ms
64 bytes from 192.168.1.3: icmp_seq=7 ttl=64 time=0.068 ms
64 bytes from 192.168.1.3: icmp_seq=8 ttl=64 time=0.062 ms
64 bytes from 192.168.1.3: icmp_seq=9 ttl=64 time=0.043 ms
64 bytes from 192.168.1.3: icmp_seq=10 ttl=64 time=0.079 ms

--- 192.168.1.3 ping statistics ---
10 packets transmitted, 10 received, 0% packet loss, time 9389ms
rtt min/avg/max/mdev = 0.043/0.064/0.114/0.021 ms
```

Iperf3 Network Bandwidth Test:

The **iperf3 bandwidth test** measures network throughput between two *containers*, where one acts as a **server** (iperf3 -s) and listens on **port** 5201, while the other operates as a **client** (iperf3 -c 192.168.1.3 -p 5201), generating traffic towards the server. This test quantifies the available **bandwidth**, providing insights into network performance under load. The results display high throughput, confirming that the internal container network is capable of handling substantial data transfer rates. Variability in results could be attributed to network stack processing, available system resources, or overlay network configurations in **Docker**.

- iperf3 -s 5201
- iperf3 -c 192.168.1.3 5201

```
root@6ee8bb165df9:/# iperf3 -c 192.168.1.3 5201
Connecting to host 192.168.1.3, port 5201
[  5] local 192.168.1.2 port 57210 connected to 192.168.1.3 port 5201
[ ID] Interval           Transfer     Bitrate         Retr  Cwnd
[  5]   0.00-1.00   sec  10.8 GBytes  92.8 Gbits/sec  306   2.52 MBytes
[  5]   1.00-2.00   sec  10.9 GBytes  93.6 Gbits/sec   72   2.52 MBytes
[  5]   2.00-3.00   sec  10.8 GBytes  92.5 Gbits/sec    3   2.52 MBytes
[  5]   3.00-4.00   sec  10.7 GBytes  91.8 Gbits/sec    0   2.52 MBytes
[  5]   4.00-5.00   sec  10.7 GBytes  92.3 Gbits/sec  329   2.53 MBytes
[  5]   5.00-6.00   sec  10.8 GBytes  93.1 Gbits/sec    1   2.53 MBytes
[  5]   6.00-7.00   sec  10.7 GBytes  92.3 Gbits/sec    1   2.53 MBytes
[  5]   7.00-8.00   sec  10.7 GBytes  91.9 Gbits/sec    0   2.54 MBytes
[  5]   8.00-9.00   sec  10.7 GBytes  91.7 Gbits/sec    1   2.55 MBytes
[  5]   9.00-10.00  sec  10.7 GBytes  91.5 Gbits/sec    3   2.55 MBytes
- - - - - - - - - - - - - - - - - - - - - - - - -
[ ID] Interval           Transfer     Bitrate         Retr
[  5]   0.00-10.00  sec   108 GBytes  92.4 Gbits/sec  716             sender
[  5]   0.00-10.04  sec   108 GBytes  92.0 Gbits/sec                  receiver

iperf Done.
```

Netcat throughput test:

The **netcat throughput test** evaluates raw data transfer performance between two containers using nc. The server container listens on port 5000 (nc -l -p 5000 > /dev/null), while the client container transmits data using dd (dd if=/dev/zero bs=1M count=100 | nc 192.168.1.3 5000). This method streams 100MB of zeroed data to the server, measuring the transfer speed in real-time. The reported speed of approximately 7.96 MB/s suggests stable transfer rates, but performance may vary based on network stack optimizations, buffer sizes, and resource contention. The netcat test is effective for assessing point-to-point data transmission efficiency in a containerized environment.

- nc -l -p 5000 > /dev/null
- time dd if=/dev/zero bs=1M count=100 | nc 192.168.1.3 5000

```
root@6ee8bb165df9:/# time dd if=/dev/zero bs=1M count=100 | nc 192.168.1.3 5000
100+0 records in
100+0 records out
104857600 bytes (105 MB, 100 MiB) copied, 0.139832 s, 750 MB/s
```

Note: before performing network tests, ensure to enter both sender and receiver container, using docker exec command.

Conclusions

The **benchmark** tests conducted on the containerized system provide valuable insights into the performance characteristics of containers compared to traditional virtual machines (**VMs**). Each test evaluates a specific aspect of computational efficiency, including high-performance Linpack (**HPL**) for floating-point operations, **CPU** stress testing using *Sysbench* and *Stress-NG*, *disk I/O* assessment via *IOzone*, and network performance analysis with tools like *ping*, *Netcat*, and *iPerf3*. These tests highlight fundamental differences between containers and VMs in terms of performance overhead, resource efficiency, and practical deployment scenarios.

The **HPL test** results demonstrate that containerized environments deliver near-native performance for floating-point computations. Since containers share the host operating system's kernel, they introduce **minimal overhead**, allowing for more **direct access to CPU resources**. This enables high computational efficiency, making containers an attractive choice for workloads demanding high floating-point throughput. In contrast, VMs operate with a hypervisor layer that introduces additional latency and resource allocation inefficiencies, which can lead to slight performance degradation in HPL benchmarks. However, VMs provide better **isolation and security**, ensuring that workloads remain **unaffected by system-wide kernel** modifications or vulnerabilities.

CPU testing using *Sysbench* and *Stress-NG* further reinforces the efficiency of containers in utilizing processor resources. Containers, due to their lightweight architecture, execute CPU-bound tasks with **minimal overhead**, leveraging the host's computational power without significant performance loss. The absence of full-fledged virtualization layers results in lower latency and faster execution times for intensive workloads. In comparison, VMs experience **additional CPU scheduling** overhead imposed by the **hypervisor**, reducing overall performance. While modern hypervisors have been optimized to minimize these inefficiencies, containers still exhibit superior responsiveness and resource utilization, particularly in environments requiring rapid scaling and real-time processing capabilities.

Disk I/O performance, evaluated using *IOzone*, illustrates another key advantage of containerized systems. Containers benefit from direct access to the host filesystem or dedicated storage volumes with minimal abstraction, ensuring high read and write speeds. The reduced virtualization overhead allows for better disk throughput and lower I/O latency. Conversely, VMs rely on virtualized disk images, which introduce **additional layers of abstraction** between the application and physical storage. This can lead to performance bottlenecks, particularly when dealing with large-scale data

operations. Although VMs offer better **isolation** and are often preferred in environments where data security is a primary concern, containers remain the preferred choice for **high-performance**, I/O-intensive applications due to their lower overhead and more efficient storage access.

Network performance testing with *ping*, *Netcat*, and *iPerf3* provides insights into communication latency and bandwidth utilization within **containerized** environments compared to **VMs**. Containers exhibit lower network latency due to their lightweight architecture and the use of high-speed virtual networking mechanisms. Communication between containers on the same host is particularly efficient, as it bypasses the need for complex virtual network translations. On the other hand, VMs introduce additional network abstraction layers, often resulting in higher latency and reduced throughput. While modern hypervisor-based networking solutions have improved VM network performance, containers still maintain an edge in scenarios requiring low-latency, high-bandwidth communication, such as distributed computing and real-time data processing.

Despite the evident performance advantages of **containers**, **VMs** retain certain strengths that make them indispensable in specific use cases. One of the most significant benefits of VMs is their robust isolation and **security** model. Because VMs include a separate guest operating system for each instance, they provide **strong isolation** between workloads, preventing issues such as *kernel-level vulnerabilities* and cross-container security risks. This makes VMs ideal for environments that require stringent security controls, such as multi-tenant cloud deployments and regulatory-compliant infrastructures. Containers, although efficient, share the host kernel, making them more susceptible to security risks if not properly managed.

Another advantage of VMs is their ability to support a wide range of operating systems and configurations. Since VMs encapsulate an entire OS, they enable deployment of applications that require different kernel versions or system-level configurations that are not easily adaptable within a containerized setup. This flexibility is particularly useful for legacy applications or scenarios where software dependencies necessitate full system virtualization. In contrast, containers are limited to the host OS kernel, which can create *compatibility challenges when dealing with diverse software environments*.

On the other hand, containers excel in **scalability**, **resource efficiency**, and deployment speed. Due to their lightweight nature, containers can be instantiated or terminated rapidly, making them highly suitable for dynamic workloads that require elastic scaling. The ability to deploy multiple containers on a single host without the overhead of multiple OS instances significantly improves resource utilization.

The disadvantage of VMs in terms of resource consumption is another factor that makes containers preferable for **high-performance** computing environments. Since each VM requires a dedicated OS instance, memory and CPU resources are allocated inefficiently compared to containers, where only the application and necessary dependencies are packaged together. This results in higher RAM and disk usage per VM, reducing the overall system efficiency, especially in large-scale deployments. Containers mitigate this issue by sharing the host OS and utilizing resources more efficiently, leading to higher density and lower operational costs.

From a management perspective, containers also simplify application deployment and orchestration. With technologies like **Docker** and **Kubernetes**, containerized workloads can be easily orchestrated, scaled, and managed across distributed infrastructures. The declarative nature of container

orchestration frameworks enables automated deployment, load balancing, and fault tolerance, which are more complex to achieve with traditional VM-based infrastructures. VMs, while benefiting from mature management tools such as VMware vSphere and OpenStack, require more overhead in provisioning, maintenance, and migration compared to containerized solutions.

Overall, the choice between containers and **VMs** depends on the specific workload requirements. Containers provide superior performance in *CPU-bound* tasks, *disk I/O*, *and network communication* due to their minimal overhead and direct access to system resources. They are particularly advantageous in scenarios requiring high scalability, fast deployment, and efficient resource utilization. However, VMs remain the preferred choice in environments where strong isolation, multi-OS support, and enhanced security are paramount. The results from the HPCC benchmarking tests reinforce the notion that while containers offer significant efficiency gains, they should be strategically deployed based on workload characteristics, security considerations, and infrastructure constraints.

References

- Cloud basic repository: https://github.com/Foundations-of-HPC/Cloud-basic-2024
- Ubuntu 24.04 server: https://ubuntu.com/download/server.
- Oracle VM VirtualBox: https://www.oracle.com/virtualization/technologies/vm/downloads/virtualbox-downloads.html
- Docker Desktop: https://www.docker.com/products/docker-desktop/
- HPC Challenge: https://hpcchallenge.org/hpcc/
- HPL tuning: https://www.netlib.org/benchmark/hpl/tuning.html
- Tweak HPL parameters: : https://www.advancedclustering.com/act_kb/tune-hpl-dat-file/
- Configure DNSMASQ: https://computingforgeeks.com/install-and-configure-dnsmasq-on-ubuntu/?expand_article=1
- Configure NFS Mount: https://www.howtoforge.com/how-to-install-nfs-server-and-client-on-ubuntu-22-04/
- SSH between containers: https://stackoverflow.com/questions/53984274/ssh-from-one-container-to-another-container
- sysbench: https://github.com/akopytov/sysbench
- stress-ng: https://manpages.ubuntu.com/manpages/bionic/man1/stress-ng.1.html
- IOzone: http://www.iozone.org/
- Iperf3: https://iperf.fr/