Cloud computing Project - Cristiano Baldassi

Cloud Basic

Virtual machines

Project Structure

```
.
├── ansible
│   └── setup.yaml
├── assignment.md
├── hpl
│   └── HPL.dat
├── libvirt
│   └── Vagrantfile
├── README.md
├── results
│   ├── libvirt
│   └── virtualbox
├── scripts
│   ├── run_test.sh
│   └── tests
│       ├── cpu_test.sh
│       ├── io_test.sh
│       ├── memory_test.sh
│       └── network_test.sh
└── virtualbox
    └── Vagrantfile
```

Description

The project utilizes a Vagrantfile and an Ansible playbook to provision a specific network environment designed for performance benchmarking, particularly focusing on high-performance computing (HPC) workloads and shared storage performance.

The environment, as defined by the `Vagrantfile`, consists of the following virtual machines:

- `server-00` **(192.168.50.10):** This VM is designated and configured by the Ansible playbook as the central **NFS (Network File System) server**. Its primary role is to export a specific directory as shared storage across the network, and it is also configured to function as an **iPerf server** for network performance testing.

- `node-00` **(192.168.50.11):** This VM acts as a **worker machine**. It is configured by the Ansible playbook as an NFS client to access the shared storage provided by `server-00`. Additionally, this node is equipped with a variety of general benchmarking tools (such as stress-ng, sysbench, iozone3, iperf) and has a complete HPC environment (including custom-compiled OpenBLAS, OpenMPI, and the HPL LINPACK benchmark) set up within the `vagrant` user's home directory.

Both VMs are based on the `generic/ubuntu2004` box and are configured with a private network.

The provisioned environment is equipped to perform a suite of performance benchmarks using the scripts located in the `scripts/` directory (as copied to `/home/vagrant/scripts/` on the worker nodes). These tests are designed to evaluate the key performance aspects of the deployed virtual machines and the network infrastructure connecting them.

The benchmark suite includes tests for:

- **CPU Performance:** These tests measure the processing capabilities of the worker nodes. They utilize `stress-ng` for various CPU-intensive workloads, including matrix operations and other computational tasks, as well as a general CPU benchmark from `sysbench`.
- **Memory Performance:** This category assesses the memory subsystem's speed and capacity. `stress-ng` is used to apply different types of memory pressure (VM, malloc, bigheap), and `sysbench` provides a dedicated memory bandwidth and latency test.
- **I/O Performance:** Focusing on storage performance, these tests evaluate how efficiently the system can handle disk read and write operations. Both `iozone` for filesystem-level testing and `fio` for more granular block-level I/O are used.
- **Network Performance:** Essential for understanding the connectivity of the nodes, these tests measure network bandwidth using `iperf` and network latency using `ping` between the worker and server nodes.

What are Virtualbox, Libvirt, KVM and QEMU?

VirtualBox is a cross-platform, type-2 hypervisor developed by Oracle that enables full system virtualization by running multiple isolated guest operating systems on a single host. It operates in user space atop a host OS and depends on the host for resource and device management.
Its modular architecture includes a virtual machine monitor (VMM), a CPU virtualization engine with support for hardware acceleration (Intel VT-x, AMD-V), and emulated components for I/O devices, storage, networking, and display.

Libvirt is a **virtualization management library and API**. It is not a hypervisor itself, but rather a **toolkit for managing hypervisors**, especially **KVM** and **QEMU**, although it can also manage Xen, LXC, and even VirtualBox.

Libvirt provides a **unified interface** to manage VMs, supporting tasks such as creating, starting, stopping, cloning, and migrating virtual machines. It includes tools like:

- `virsh` – command-line utility
- `virt-manager` – graphical front-end
- `virt-install` – for provisioning new VMs

KVM is a **type-1 (bare-metal) hypervisor** built into the Linux kernel. It turns the Linux operating system into a full-fledged hypervisor, allowing you to run multiple virtual machines (VMs) on a physical host, each with its own virtual CPU, memory, disk, and network interfaces.

KVM uses **hardware-assisted virtualization** (Intel VT-x or AMD-V) to achieve near-native performance. Each VM runs as a regular Linux process, but with special kernel modules (`kvm` and `kvm-intel` or `kvm-amd`) that handle low-level virtualization tasks.

QEMU (Quick EMUlator) is an open-source machine emulator and virtualizer. It can emulate entire hardware systems or individual programs compiled for different CPU architectures. When combined with **KVM** on Linux, it enables high-performance virtualization by leveraging hardware acceleration (Intel VT-x, AMD-V).

Vagrant

What is Vagrant?

Vagrant is an open-source tool that simplifies the creation and management of portable virtual development environments. It provides a consistent workflow to set up and configure virtual machines using a declarative configuration file called a **Vagrantfile**.

Vagrant works by using **boxes**, which are pre-packaged base images for virtual machines, and interacts with various **providers** like VirtualBox, VMware, and Libvirt. After a machine is created, **provisioners** automatically install and configure software based on the Vagrantfile. This process ensures reproducible, portable, and consistent development environments across different machines and team members.

Vagrantfile

1. **Variable Definitions:**

```
VAGRANT_BOX = "generic/ubuntu2004"
VM_MEMORY = 4096 # Default memory
VM_CPUS = 4     # Default CPUs

servers = [
  { :hostname => "server-00", :ip => "192.168.50.10" },
  { :hostname => "node-00", :ip => "192.168.50.11" },
```

```
# { :hostname => "node-01", :ip => "192.168.50.12" },
]
```

- `VAGRANT_BOX = "generic/ubuntu2004"` : This line defines a variable `VAGRANT_BOX` and sets its value to `"generic/ubuntu2004"` . This specifies the base operating system image (the "box") that Vagrant will use to create the VMs
- `VM_MEMORY = 4096` : Sets a default variable `VM_MEMORY` to 4096.
- `VM_CPUS = 4` : Sets a default variable `VM_CPUS` to 4.
- `servers = [...]` : This defines an array named `servers` . Each element in the array is a hash representing a virtual machine to be created. Each hash contains a `:hostname` and an `:ip` address for the respective machine.

2. **Vagrant Configuration Block:**

```
Vagrant.configure("2") do |config|
  # ... configuration details ...
end
```

- `Vagrant.configure("2") do |config| ... end` : This is the main configuration block for Vagrant, using version "2" of the configuration format. All subsequent configuration is done within this block, using the `config` object.

3. **Base Box Configuration:**

```
  config.vm.box = VAGRANT_BOX
```

- `config.vm.box = VAGRANT_BOX` : This line sets the base box for *all* VMs defined in this Vagrantfile to the value stored in the `VAGRANT_BOX` variable.

4. **Provider-Specific Configuration (libvirt):**

```
  config.vm.provider "libvirt" do |lv|
    lv.memory = VM_MEMORY
    lv.cpus = VM_CPUS
    lv.qemu_use_session = false
  end
```

- `config.vm.provider "libvirt" do |lv| ... end` : This block contains configuration specific to the `libvirt` provider. The `lv` object is used to access libvirt-specific settings.
- `lv.memory = VM_MEMORY` : Sets the memory for the VMs using the `VM_MEMORY` variable (4096 MB).
- `lv.cpus = VM_CPUS` : Sets the number of CPUs for the VMs using the `VM_CPUS` variable (4).

- `lv.qemu_use_session = false`: Configures libvirt to use the system QEMU instance rather than a user session instance.

5. **Provisioning with Ansible:**

```
config.vm.provision "ansible" do |ansible|
  ansible.playbook = "../ansible/setup.yaml"
  # Define Ansible groups
  ansible.groups = {
    "server" => ["server-00"],
    "worker" => ["node-00"],
  }
end
```

- `config.vm.provision "ansible" do |ansible| ... end`: This block configures Vagrant to use Ansible for provisioning the VMs after they are created and booted. Provisioning is the process of automatically installing software, configuring settings, etc., on the VMs.
- `ansible.playbook = "../ansible/setup.yaml"`: Specifies the path to the Ansible playbook that will be executed on the VMs.
- `ansible.groups = { ... }`: This defines Ansible groups. Vagrant will automatically generate an Ansible inventory file based on the VMs it creates, and these groups will be included in that inventory. This allows the Ansible playbook (`setup.yaml`) to target specific sets of machines (e.g., tasks for the "server" group will only run on "server-00").

6. **Defining Individual Machines:**

```
servers.each do |conf|
  config.vm.define conf[:hostname] do |node|
    node.vm.hostname = conf[:hostname]
    node.vm.network "private_network", ip:conf[:ip]

    # Customize resources for the 'node-control' server
    if conf[:hostname] == "node-control"
      node.vm.provider "libvirt" do |lv|
        lv.memory = 2048 # 2GB
        lv.cpus = 2
      end
    end
  end
end
```

- `servers.each do |conf| ... end`: This loop iterates through each hash in the `servers` array defined at the beginning of the file. For each hash, the current server's configuration is available in the `conf` variable.

- `config.vm.define conf[:hostname] do |node| ... end`: Inside the loop, `config.vm.define` is used to define an individual virtual machine. The name of the machine in Vagrant is set to the `:hostname` value from the current `conf` hash (e.g., "server-00", "node-00"). The configuration for this specific machine is done using the `node` object.
- `node.vm.hostname = conf[:hostname]`: Sets the internal hostname of the VM to the value from the `conf` hash.
- `node.vm.network "private_network", ip:conf[:ip]`: Configures a private network for the VM. This creates a host-only network interface, allowing the host machine and other VMs on the same private network to communicate with this VM using the specified static IP address from the `conf` hash (e.g., 192.168.50.10, 192.168.50.11).
- `if conf[:hostname] == "node-control" ... end`: This is a conditional block. It checks if the current machine's hostname is exactly `"node-control"`. If it is, it applies specific provider settings *only* to that machine.
- `node.vm.provider "libvirt" do |lv| ... end`: This nested provider block allows you to override the default libvirt settings defined earlier for a specific machine.
- `lv.memory = 2048`: If the hostname is "node-control", its memory is set to 2048 MB (2GB), overriding the default 4096 MB.
- lv.cpus = 2: If the hostname is "node-control", its number of CPUs is set to 2, overriding the default 4.

Ansible

What is Ansible?

Ansible is an open-source IT automation tool for configuration management, application deployment, and orchestration. It's known for being **agentless**, connecting to machines via SSH. Key components include:

- **Inventory:** Lists the servers and devices to manage.
- **Modules:** Small programs executed on managed nodes to perform specific tasks.
- **Tasks:** Define a single action to be performed.
- **Playbooks:** YAML files that contain ordered lists of tasks to automate workflows.
- **Roles:** Organize playbooks and related files for reusability.

Provisioning for server node

1. **Play Definition:**

```
- name: Configure NFS and iperf server
  hosts: server
```

```
  become: yes
```

- `- name: Configure NFS and iperf server` : This line provides a descriptive name for the play. This name is shown in the output when Ansible runs the playbook, making it easier to understand what the play is doing.
- `hosts: server` : This specifies which hosts from the Ansible inventory this play should run on. In this case, it will run on all hosts that are part of the `server` group. Based on the Vagrantfile we discussed earlier, this would target the `server-00` VM.
- `become: yes` : This indicates that Ansible should execute the tasks in this play with elevated privileges (like `sudo`).

2. **Variable Definitions:**

```
  vars:
    nfs_share_path: "/mnt/nfs_share"
    allowed_client_ip: "192.168.50.10"
```

- `vars:` : This section defines variables that can be used within this play.
- `nfs_share_path: "/mnt/nfs_share"` : Defines a variable `nfs_share_path` and sets its value to `/mnt/nfs_share` . This path will be used for the directory that will be shared via NFS.
- `allowed_client_ip: "192.168.50.10"` : Defines a variable `allowed_client_ip` and sets its value to `"192.168.50.10"` . This IP address will be configured in the NFS exports file to grant access to a specific client.

3. **Tasks:**

```
  tasks:
    - name: task_name
      ansible_module:
        ...
```

- `tasks:` : This section lists the tasks that Ansible will execute sequentially on the target hosts.

```
- name: Update apt package cache
  apt:
    update_cache: yes
```

Update apt package cache: uses the apt module to manage packages and is configured with `update_cache: yes` to ensure the local package cache is refreshed before installing any packages.

```
- name: Install iperf server
   apt:
     name: iperf
     state: present
```

Install iperf server: uses the `apt` module to manage packages and is configured with `name: iperf` to specify the package and `state: present` to ensure it is installed.

```
- name: Create iperf server systemd service file
   copy:
     dest: /etc/systemd/system/iperf-server.service
     content: |
       [Unit]
       Description=iPerf Server Daemon
       After=network.target

       [Service]
       ExecStart=/usr/bin/iperf --server --daemon
       Restart=on-failure

       [Install]
       WantedBy=multi-user.target
     mode: '0644'
     owner: root
     group: root
```

Create iperf server systemd service file: uses the `copy` module to create a file at `/etc/systemd/system/iperf-server.service` with the specified content, setting the mode to `0644` and the owner and group to `root`.

```
- name: Reload systemd and start/enable iperf server service
   systemd_service:
     name: iperf-server
     state: started
     enabled: yes
     daemon_reload: yes
```

Reload systemd and start/enable iperf server service: uses the `systemd_service` module to manage the iperf-server service, ensuring its state is started, it is enabled to start on boot, and `daemon_reload: yes` is used to pick up the new service file.

```
- name: Install NFS kernel server
   apt:
     name: nfs-kernel-server
     state: present
```

Install NFS kernel server: uses the `apt` module to manage packages and is configured with `name: nfs-kernel-server` to specify the package and `state: present` to ensure it is installed.

```
- name: Create NFS share directory
   file:
     path: "{{ nfs_share_path }}" # Using the variable here
     state: directory
     mode: '0777'
```

Create NFS share directory: uses the `file` module to manage files and directories and is configured with path: `{{ nfs_share_path }}` to specify the directory using a variable, `state: directory` to ensure it is a directory, and mode: `0777` to set permissions.

```
- name: Add NFS export configuration to /etc/exports
   lineinfile:
     path: /etc/exports
     line: "{{ nfs_share_path }} {{ allowed_client_ip }}
(rw,sync,no_subtree_check)" # Using variables here
     create: yes # Create the file if it doesn't exist
     owner: root
     group: root
     mode: '0644'
```

Add NFS export configuration to /etc/exports: uses the `lineinfile` module to ensure a specific line is present in the file at `path: /etc/exports`, using variables for the share path and client IP, and is configured with `create: yes`, `owner: root`, `group: root`, and `mode: '0644'`.

```
- name: Export NFS shares
   command: exportfs -a
   notify: Restart NFS server
```

Export NFS shares: uses the `command` module to execute the `exportfs -a` command, which exports the NFS shares defined in `/etc/exports`. It also uses `notify: Restart NFS` server to trigger the specified handler if the command makes a change.

```
- name: Ensure NFS kernel server is running and enabled
   systemd:
     name: nfs-kernel-server
     state: started
     enabled: yes
```

Ensure NFS kernel server is running and enabled: uses the `systemd` module to manage the nfs-kernel-server service, ensuring its state is started and it is enabled to start on

boot.

4. **Handlers:**

```
  handlers:
    - name: Restart NFS server
      systemd:
        name: nfs-kernel-server
        state: restarted
```

- `handlers:` : This section defines handlers, which are tasks that are only run when explicitly notified by a task.

```
- name: Restart NFS server
   systemd:
     name: nfs-kernel-server
     state: restarted
```

Restart NFS server: uses the `systemd` module to manage the nfs-kernel-server service and is configured with state: restarted to restart the service.

Provisioning for worker node

1. **Play: Install benchmarking tools**

```
- name: Install benchmarking tools
  hosts: worker
  become: yes
```

Tasks:

```
- name: Update apt package list
   apt:
     update_cache: yes
```

Update apt package list: uses the `apt` module to manage packages and is configured with `update_cache: yes` to ensure the local package cache is refreshed before installing any packages.

```
- name: Install benchmarking tools
   apt:
     name:
       - stress-ng
       - sysbench
       - iozone3
```

```
        - iperf
        - fio
        - iputils-ping
        - vim
      state: present
```

Install benchmarking tools: uses the `apt` module to manage packages and is configured with `name:` listing multiple packages and `state: present` to ensure all listed packages are installed.

```
- name: Copy benchmarking scripts to the worker VMs
   copy:
     src: ../scripts/ # Path to your scripts directory on the Ansible
control machine
     dest: /home/vagrant/scripts/ # Destination path on the worker VMs
     owner: vagrant # Optional: Set the owner of the copied files
     group: vagrant # Optional: Set the group of the copied files
     mode: '0755' # Optional: Set permissions for the copied scripts
(e.g., make them executable)
```

Copy benchmarking scripts to the worker VMs: uses the `copy` module to copy files or directories from the Ansible control machine to the remote host. It is configured with `src: ../scripts/` specifying the source directory, `dest: /home/vagrant/scripts/` for the destination path on the worker VMs, and optional parameters for owner, group, and mode.

2. **Play: Configure NFS client**

```
- name: Configure NFS client
  hosts: worker
  become: yes
```

- `- name: Configure NFS client` : name for this play.
- `hosts: worker` : Specifies that this play should run on all hosts in the `worker` group.
- `become: yes` : Indicates that tasks in this play require elevated privileges (`sudo`).
- `vars:` : Defines variables specific to this play for the NFS configuration.

Tasks:

```
- name: Update apt package cache
   apt:
     update_cache: yes
```

Update apt package cache: uses the `apt` module to manage packages and is configured with `update_cache: yes` to ensure the local package cache is refreshed before installing any packages.

```
- name: Install NFS client package (nfs-common)
   apt:
     name: nfs-common
     state: present
```

Install NFS client package (nfs-common): uses the `apt` module to manage packages and is configured with name: nfs-common to specify the NFS client package and state: present to ensure it is installed.

```
- name: Create local mount point directory
   file:
     path: "{{ local_mount_point }}"
     state: directory
     mode: '0755' # Standard directory permissions
```

Create local mount point directory: uses the `file` module to manage files and directories and is configured with `path: "{{ local_mount_point }}"` to specify the directory using a variable, `state: directory` to ensure it is a directory, and `mode: '0755'` to set permissions.

```
- name: Mount the NFS share and add to /etc/fstab
   mount:
     src: "{{ nfs_server_address }}:{{ remote_nfs_share }}"
     path: "{{ local_mount_point }}"
     fstype: nfs
     opts: "defaults,_netdev" # Options including _netdev for network
mounts
     state: mounted # Ensures it's mounted and adds/updates the fstab
entry
```

Mount the NFS share and add to /etc/fstab: uses the `mount` module to manage filesystem mounts. It is configured with `src: "{{ nfs_server_address }}:{{ remote_nfs_share }}"` specifying the remote NFS share using variables, `path: "{{ local_mount_point }}"` for the local mount point, `fstype: nfs` to specify the filesystem type, `opts: "defaults,_netdev"` for mount options, and `state: mounted` to ensure the share is mounted and an entry is added/updated in `/etc/fstab`.

3. **Play: Install and Configure HPL LINPACK**

```
- name: Install and Configure HPL LINPACK
  hosts: worker
  become: yes
```

```
  vars:
    # Force installation under the vagrant user's home
    actual_user: vagrant
    user_home: "/home/{{ actual_user }}"
    openblas_version: "v0.3.21"
    openmpi_version: "4.1.4"
    hpl_version: "2.3"
    openblas_dir: "{{ user_home }}/opt/OpenBLAS"
    openmpi_dir: "{{ user_home }}/opt/OpenMPI"
    hpl_dir: "{{ user_home }}/hpl"
    parallel_jobs: 4
```

- `- name: Install and Configure HPL LINPACK` : The name for this play
- `hosts: worker` : Specifies that this play will be executed on all hosts belonging to the `worker` group in the Ansible inventory.
- `become: yes` : Indicates that the tasks within this play require root privileges (`sudo`)
- `vars:` : Defines variables used throughout this play to manage paths, versions, and user details for the installation process.

Tasks:

```
    - name: Update apt cache
      apt:
        update_cache: yes
```

Update apt cache: This task uses the `apt` module to interact with the APT package manager. `update_cache: yes` ensures that the local package list on the target hosts is updated from the repositories before any installation attempts.

```
    - name: Install build dependencies
      apt:
        name:
          - build-essential
          - hwloc
          - libhwloc-dev
          - libevent-dev
          - gfortran
          - git
          - wget
        state: present
```

Install build dependencies: This task uses the `apt` module to install a list of necessary packages (`build-essential` , `hwloc` , etc.) required for compiling and building OpenBLAS, OpenMPI, and HPL. `state: present` ensures that these packages are installed if they are not already.

```
- name: Ensure vagrant opt directory exists
      file:
        path: "{{ user_home }}/opt"
        state: directory
        owner: vagrant
        group: vagrant
        mode: '0755'
      become_user: vagrant
```

Ensure vagrant opt directory exists: This task uses the `file` module to manage the filesystem. It ensures that the directory specified by `path: "{{ user_home }}/opt"` exists (`state: directory`). It also sets the `owner` and `group` to `vagrant` and the directory permissions (`mode: '0755'`). `become_user: vagrant` ensures this task is executed as the `vagrant` user, not root, despite `become: yes` being set at the play level.

```
    - name: Clone OpenBLAS repository
      git:
        repo: https://github.com/xianyi/OpenBLAS.git
        dest: "{{ user_home }}/OpenBLAS"
        version: "{{ openblas_version }}"
        update: no
      become_user: vagrant
```

Clone OpenBLAS repository: This task uses the `git` module to manage Git repositories. It clones the specified OpenBLAS repository (`repo`) into the destination path (`dest`) within the `vagrant` user's home directory. `version: "{{ openblas_version }}"` checks out a specific tag or branch, and `update: no` prevents updating the repository if it already exists. `become_user: vagrant` ensures the clone operation is performed as the `vagrant` user.

```
    - name: Compile OpenBLAS
      become_user: vagrant
      shell: |
        cd {{ user_home }}/OpenBLAS
        make -j {{ parallel_jobs }}
      args:
        chdir: "{{ user_home }}/OpenBLAS"
```

Compile OpenBLAS: This task uses the `shell` module to run shell commands. It changes the directory to the OpenBLAS source directory and then runs the `make` command with the `-j` option to compile using a specified number of parallel jobs (`parallel_jobs`). `become_user: vagrant` ensures the compilation is done as the `vagrant` user. The `args: chdir` parameter provides an alternative way to specify the working directory for the command.

```
    - name: Install OpenBLAS into vagrant opt
      become_user: vagrant
      shell: |
        cd {{ user_home }}/OpenBLAS
        make PREFIX={{ openblas_dir }} install
```

Install OpenBLAS into vagrant opt: This task uses the `shell` module to run shell commands. It changes the directory to the OpenBLAS source and runs `make install` with the `PREFIX` variable set to the desired installation directory (`openblas_dir`), placing the compiled libraries and headers in that location. `become_user: vagrant` ensures the installation is performed as the `vagrant` user.

```
    - name: Download OpenMPI tarball
      become_user: vagrant
      get_url:
        url: "https://download.open-mpi.org/release/open-
mpi/v4.1/openmpi-{{ openmpi_version }}.tar.gz"
        dest: "{{ user_home }}/openmpi-{{ openmpi_version }}.tar.gz"
```

Download OpenMPI tarball: This task uses the `get_url` module to download a file from a specified URL (`url`) to a destination path (`dest`) on the target host within the `vagrant` user's home. `become_user: vagrant` ensures the download is done as the `vagrant` user.

```
    - name: Extract OpenMPI
      become_user: vagrant
      unarchive:
        src: "{{ user_home }}/openmpi-{{ openmpi_version }}.tar.gz"
        dest: "{{ user_home }}"
        remote_src: yes
```

Extract OpenMPI: This task uses the `unarchive` module to extract a compressed archive. It extracts the OpenMPI tarball (`src`) into the specified destination directory (`dest`). `remote_src: yes` indicates that the source file is on the remote (target) host. `become_user: vagrant` ensures the extraction is performed as the `vagrant` user.

```
    - name: Configure OpenMPI
      become_user: vagrant
      shell: |
        cd {{ user_home }}/openmpi-{{ openmpi_version }}
        CFLAGS="-Ofast -march=native" ./configure --prefix={{
openmpi_dir }}
      args:
        chdir: "{{ user_home }}/openmpi-{{ openmpi_version }}"
```

Configure OpenMPI: This task uses the `shell` module to run shell commands. It changes the directory to the extracted OpenMPI source and runs the `./configure` script with specific `CFLAGS` for optimization and sets the installation prefix (`--prefix`) to the `openmpi_dir` variable. `become_user: vagrant` ensures the configuration is done as the `vagrant` user. The `args: chdir` parameter specifies the working directory for the command.

```
    - name: Compile OpenMPI
      become_user: vagrant
      shell: |
        cd {{ user_home }}/openmpi-{{ openmpi_version }}
        make -j {{ parallel_jobs }}
      args:
        chdir: "{{ user_home }}/openmpi-{{ openmpi_version }}"
```

Compile OpenMPI: This task uses the `shell` module to run shell commands. It changes the directory to the OpenMPI source and runs the `make` command with the `-j` option to compile using a specified number of parallel jobs (`parallel_jobs`). `become_user: vagrant` ensures the compilation is done as the `vagrant` user. The `args: chdir` parameter specifies the working directory for the command.

```
    - name: Install OpenMPI into vagrant opt
      become_user: vagrant
      shell: |
        cd {{ user_home }}/openmpi-{{ openmpi_version }}
        make install
```

Install OpenMPI into vagrant opt: This task uses the `shell` module to run shell commands. It changes the directory to the OpenMPI source and runs the `make install` command to install the compiled OpenMPI libraries and binaries into the directory specified during the configuration step. `become_user: vagrant` ensures the installation is performed as the `vagrant` user.

```
    - name: Add OpenMPI env to vagrant .bashrc
      become_user: vagrant
      lineinfile:
        path: "{{ user_home }}/.bashrc"
        create: yes
        line: "{{ item }}"
        state: present
      with_items:
        - 'export MPI_HOME={{ openmpi_dir }}'
        - 'export PATH=$MPI_HOME/bin:$PATH'
        - 'export LD_LIBRARY_PATH=$MPI_HOME/lib:$LD_LIBRARY_PATH'
```

Add OpenMPI env to vagrant .bashrc: This task uses the `lineinfile` module to manage lines in a file. It ensures that the specified environment variables (`MPI_HOME` , `PATH` , `LD_LIBRARY_PATH`) pointing to the OpenMPI installation directory are present in the `vagrant` user's `.bashrc` file. `create: yes` will create the file if it doesn't exist, and `state: present` ensures the lines are added if they are not already there. `with_items` iterates over the list of environment variable lines to add. `become_user: vagrant` ensures the modification is done as the `vagrant` user.

```
    - name: Download HPL source
      become_user: vagrant
      get_url:
        url: "https://netlib.org/benchmark/hpl/hpl-{{ hpl_version
}}.tar.gz"
        dest: "{{ user_home }}/hpl-{{ hpl_version }}.tar.gz"
```

Download HPL source: This task uses the `get_url` module to download the HPL source tarball from the specified URL (`url`) to the destination path (`dest`) within the `vagrant` user's home directory. `become_user: vagrant` ensures the download is done as the `vagrant` user.

```
    - name: Extract HPL
      become_user: vagrant
      unarchive:
        src: "{{ user_home }}/hpl-{{ hpl_version }}.tar.gz"
        dest: "{{ user_home }}"
        remote_src: yes
```

Extract HPL: This task uses the `unarchive` module to extract the downloaded HPL tarball (`src`) into the specified destination directory (`dest`). `remote_src: yes` indicates the source file is on the remote host. `become_user: vagrant` ensures the extraction is performed as the `vagrant` user.

```
    - name: Rename HPL directory
      become_user: vagrant
      shell: |
        if [ -d "{{ hpl_dir }}" ]; then rm -rf {{ hpl_dir }}; fi
        mv {{ user_home }}/hpl-{{ hpl_version }} {{ hpl_dir }}
```

Rename HPL directory: This task uses the `shell` module to run shell commands. It first checks if the target HPL directory (`hpl_dir`) exists and removes it if it does, then renames the extracted HPL source directory to the desired `hpl_dir` path. `become_user: vagrant` ensures these operations are performed as the `vagrant` user.

```
    - name: Create generic Makefile
      become_user: vagrant
      shell: |
        cd {{ hpl_dir }}/setup
        sh make_generic
        cp Make.UNKNOWN ../Make.linux
```

Create generic Makefile: This task uses the `shell` module to run shell commands. It changes the directory to the `setup` subdirectory within the HPL source, runs the `make_generic` script to create a generic Makefile template, and then copies `Make.UNKNOWN` to `Make.linux` in the parent HPL directory, preparing it for configuration. `become_user: vagrant` ensures these steps are performed as the `vagrant` user.

```
    - name: Configure HPL Makefile
      become_user: vagrant
      copy:
        dest: "{{ hpl_dir }}/Make.linux"
        content: |
          # HPL Makefile for {{ ansible_hostname }} (vagrant)
          SHELL        = /bin/sh
          CD           = cd
          CP           = cp
          LN_S         = ln -s
          MKDIR        = mkdir
          RM           = /bin/rm -f
          TOUCH        = touch
          ARCH         = linux
          TOPdir       = $(HOME)/hpl
          INCdir       = $(TOPdir)/include
          BINdir       = $(TOPdir)/bin/$(ARCH)
          LIBdir       = $(TOPdir)/lib/$(ARCH)
          HPLlib       = $(LIBdir)/libhpl.a

          # MPI
          MPdir        = {{ openmpi_dir }}
          MPinc        = -I$(MPdir)/include
          MPlib        = $(MPdir)/lib/libmpi.so

          # Linear Algebra
          LAdir        = {{ openblas_dir }}
          LAinc        =
          LAlib        = $(LAdir)/lib/libopenblas.a

          F2CDEFS      = -DAdd_ -DF77_INTEGER=int -DStringSunStyle
          HPL_INCLUDES = -I$(INCdir) -I$(INCdir)/$(ARCH) $(LAinc)
$(MPinc)
          HPL_LIBS     = $(HPLlib) $(LAlib) $(MPlib) -lm
```

```
            # Compiler settings
            HPL_DEFS     = $(F2CDEFS) $(HPL_INCLUDES)
            CC           = mpicc
            CCNOOPT      = $(HPL_DEFS)
            CCFLAGS      = $(HPL_DEFS) -fomit-frame-pointer -O3 -funroll-
loops -W -Wall
            LINKER       = mpif77
            LINKFLAGS    = $(CCFLAGS)
            ARCHIVER     = ar
            ARFLAGS      = r
            RANLIB       = echo
```

Configure HPL Makefile: uses the `copy` module to create or replace the file at `dest: "{{ hpl_dir }}/Make.linux"` with the specified content, which is a custom Makefile tailored for compiling HPL with OpenMPI and OpenBLAS, using variables for installation paths. It is run with `become_user: vagrant`.

```
- name: Compile HPL
   become_user: vagrant
   shell: |
     export MPI_HOME={{ openmpi_dir }}
     export PATH=$MPI_HOME/bin:$PATH
     export LD_LIBRARY_PATH=$MPI_HOME/lib:$LD_LIBRARY_PATH
     cd {{ hpl_dir }}
     make arch=linux
   args:
     chdir: "{{ hpl_dir }}"
   register: compile_result
```

Compile HPL: uses the `shell` module to execute a series of commands within a shell. It is configured with `become_user: vagrant` to run as the vagrant user, sets environment variables for OpenMPI, changes directory to the HPL source using `args: chdir`, and runs the `make arch=linux` command to compile HPL. The result is registered in the `compile_result` variable.

```
- name: Verify HPL binary exists
   stat:
     path: "{{ hpl_dir }}/bin/linux/xhpl"
   register: hpl_binary
```

Verify HPL binary exists: uses the `stat` module to get information about a file. It is configured with `path: "{{ hpl_dir }}/bin/linux/xhpl"` to check for the existence of the compiled HPL binary and registers the file's status in the `hpl_binary` variable.

```
- name: Ensure vagrant owns installation directories
   file:
```

```
      path: "{{ item }}"
      owner: vagrant
      group: vagrant
      recurse: yes
    loop:
      - "{{ openblas_dir }}"
      - "{{ openmpi_dir }}"
      - "{{ hpl_dir }}"
```

Ensure vagrant owns installation directories: uses the `file` module to manage files and directories. It is configured with `path: "{{ item }}"` which iterates through the list provided in the loop (the OpenBLAS, OpenMPI, and HPL installation directories), setting the owner and group to vagrant and applying recursively with `recurse: yes`.

```
- name: Copy a HPL.dat in to the VM
   copy:
     src: ../hpl/HPL.dat
     dest: /home/vagrant/hpl/bin/linux/HPL.dat
     owner: vagrant
     group: vagrant
     mode: '0644'
```

Copy a HPL.dat in to the VM: uses the `copy` module to copy the `HPL.dat` file from the Ansible control machine (`src: ../hpl/HPL.dat`) to the HPL binary directory on the worker VM (`dest: /home/vagrant/hpl/bin/linux/HPL.dat`), setting the `owner`, `group`, and `mode`.

```
- name: Show completion message
   debug:
     msg:
       - "HPL LINPACK has been successfully compiled for user
'vagrant'."
       - "The binary is located at: {{ hpl_dir }}/bin/linux/xhpl"
       - "To use OpenMPI and OpenBLAS, open a new terminal or run
'source ~/.bashrc' as vagrant"
   when: hpl_binary.stat.exists
```

Show completion message: uses the `debug` module to print messages to the console during playbook execution. It is configured with `msg:` listing multiple lines of text to display, including the location of the HPL binary and instructions for the user. It uses a when: `hpl_binary.stat.exists` condition to only run this task if the HPL binary was successfully found by the previous stat task.

Managing the cluster

Start the cluster:

```
> cd libvirt
> vagrant up --provider=libvirt --no-parallel
```

SSH into the vm

```
> vagrant ssh node-00
```

Halt the VMs

```
> vagrant halt
```

Destroy the VMs

```
> vagrant destroy
```

Benchmarking

Run the test suit

```
vagrant@node-00:~$ cd scripts/
vagrant@node-00:~/scripts/$ ./run_test.sh
```

Run hpl test

```
vagrant@node-00:~$ cd hpl/bin/linux/
vagrant@node-00:~/hpl/bin/linux/$ mpirun -np 4 ./xhpl 2>&1 | tee -a
~/results/hpl-test.log
```

Retrieve the results

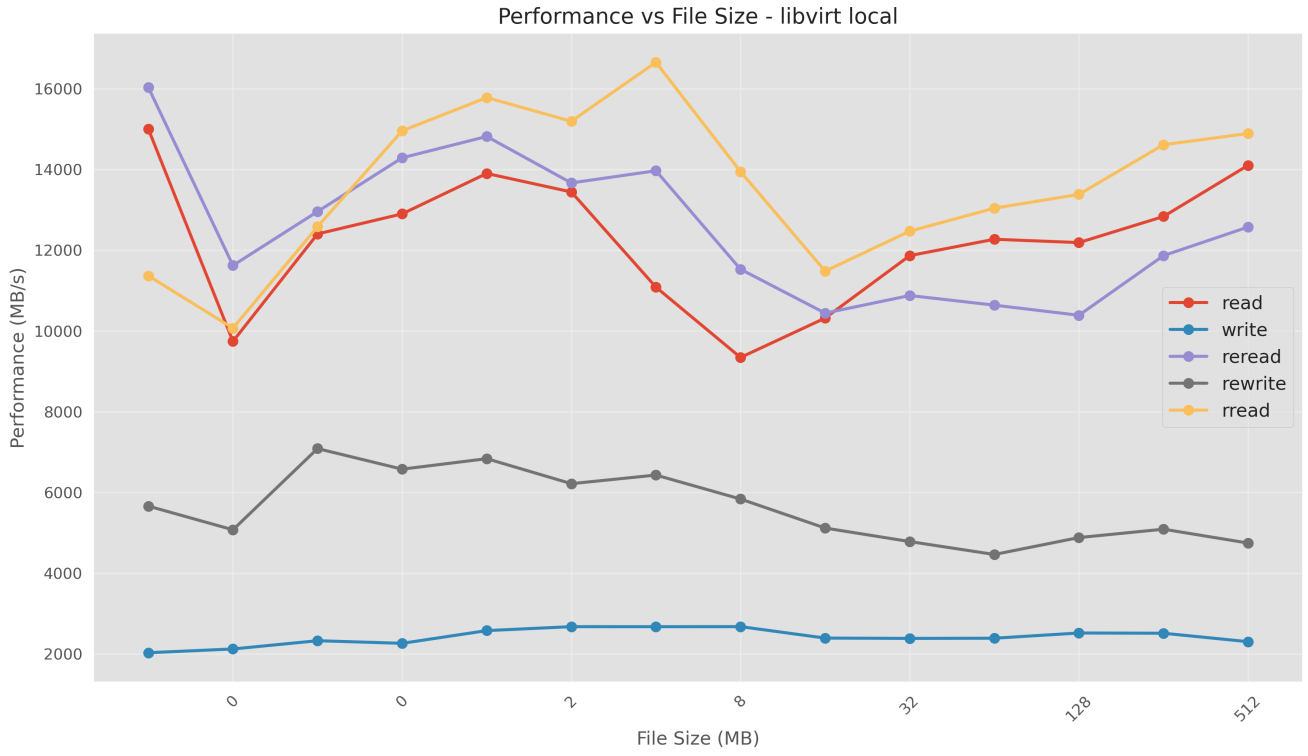
```
vagrant@node-00:~$ logout
> vagrant scp node-00:/home/vagrant/results ../results/libvirt
```

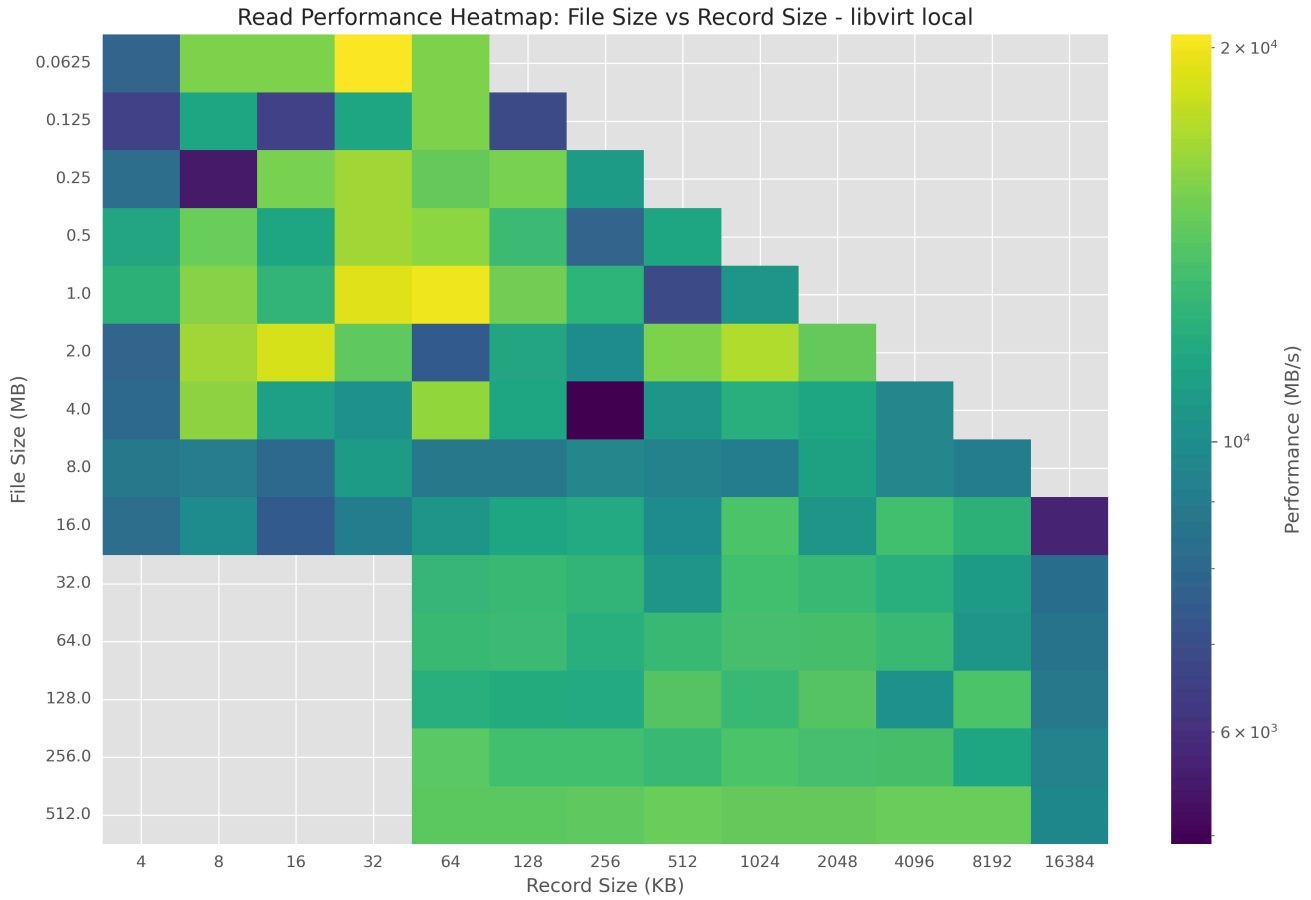
Results

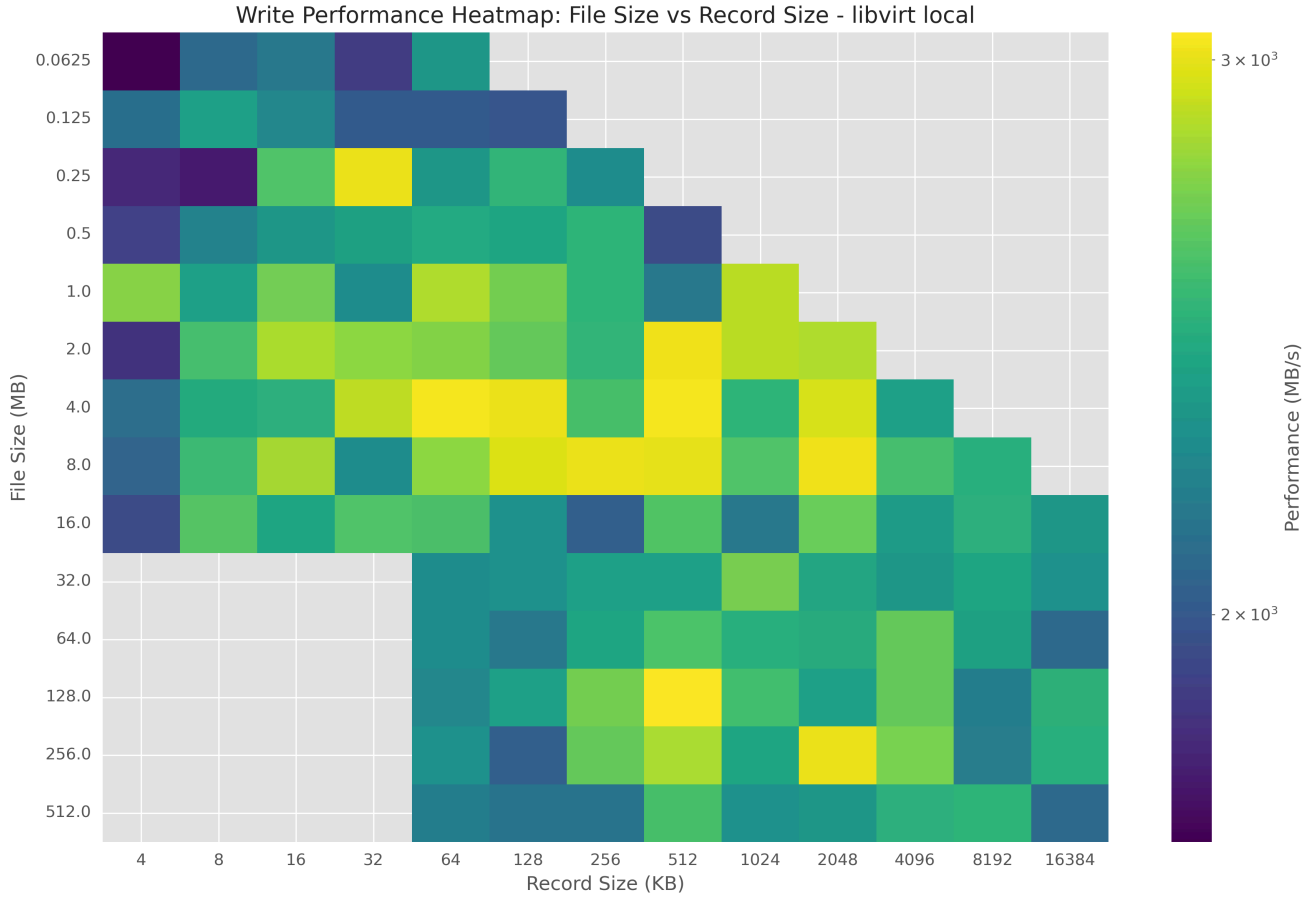
Libvirt

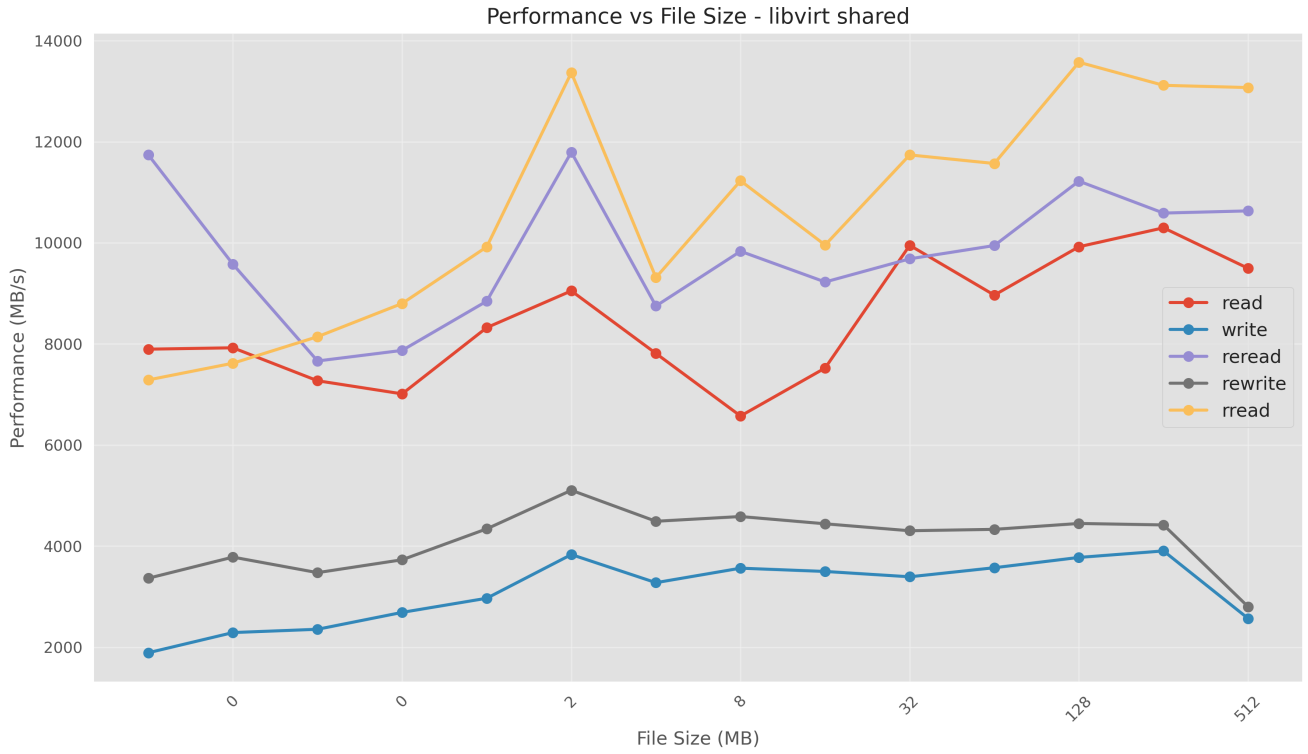
Test Type	Specific Test	Metric	Value	Units
CPU	Standard CPU Stress Test	Bogo Ops/s (real time)	789.64	bogo ops/s

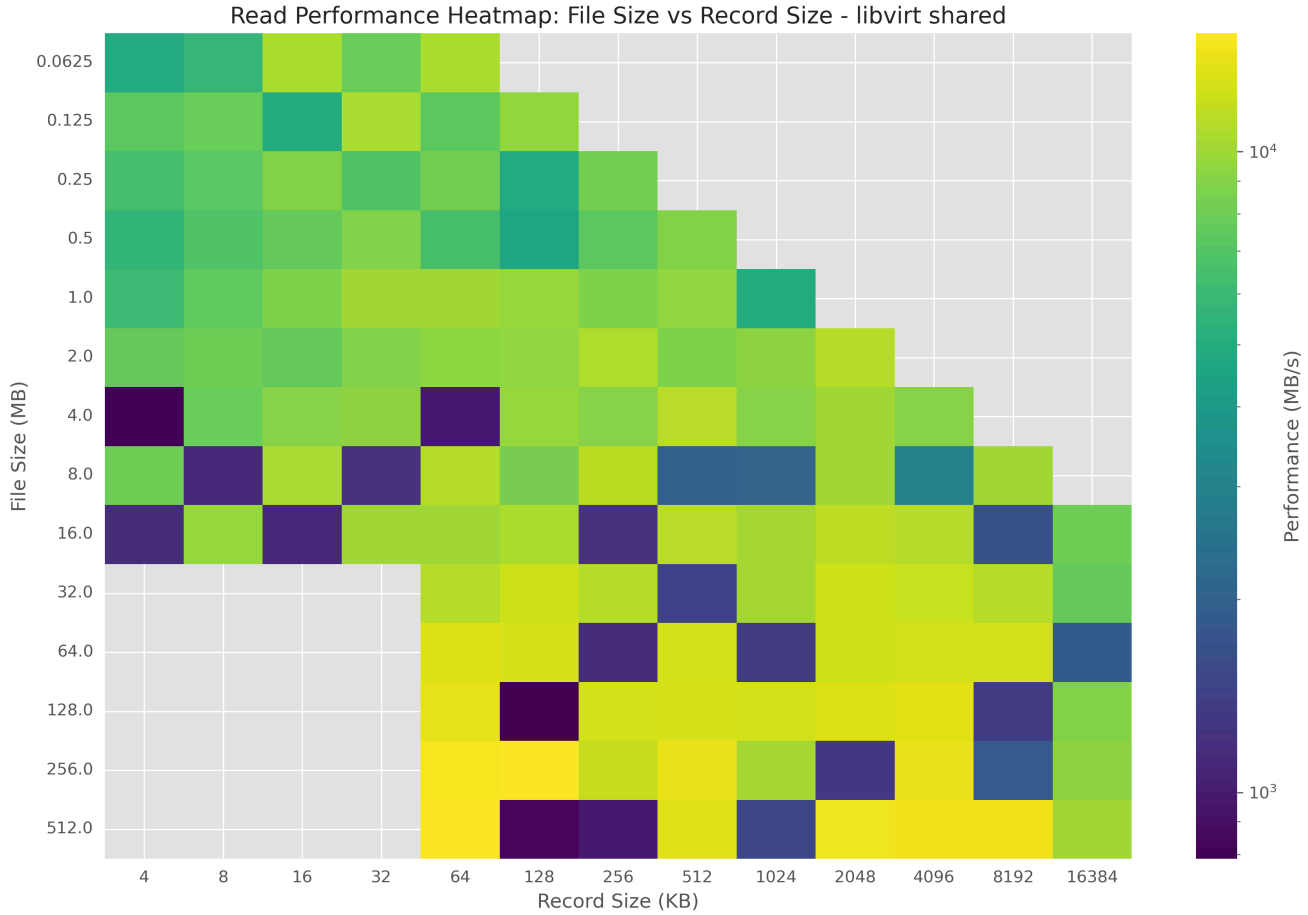
Test Type	Specific Test	Metric	Value	Units
	Matrix Multiplication	Bogo Ops/s (real time)	466.08	bogo ops/s
	FFT CPU Test	Bogo Ops/s (real time)	5000.93	bogo ops/s
	Phi CPU Test	Bogo Ops/s (real time)	97510806.98	bogo ops/s
	Sysbench CPU Test	Events per second	4978.84	events/sec
Memory	VM Stress Test	Bogo Ops/s (real time)	71165.18	bogo ops/s
	Malloc Stress Test	Bogo Ops/s (real time)	45187.48	bogo ops/s
	Bigheap Stress Test	Bogo Ops/s (real time)	20920.41	bogo ops/s
	Sysbench Memory Test	Transfer Speed	74733.10	MiB/sec
Network	iperf Bandwidth Test	Bandwidth	12.7	Gbits/sec
	ping Latency Test	RTT (avg)	0.615	ms
I/O	fio write Test	Write BW	167	MiB/s

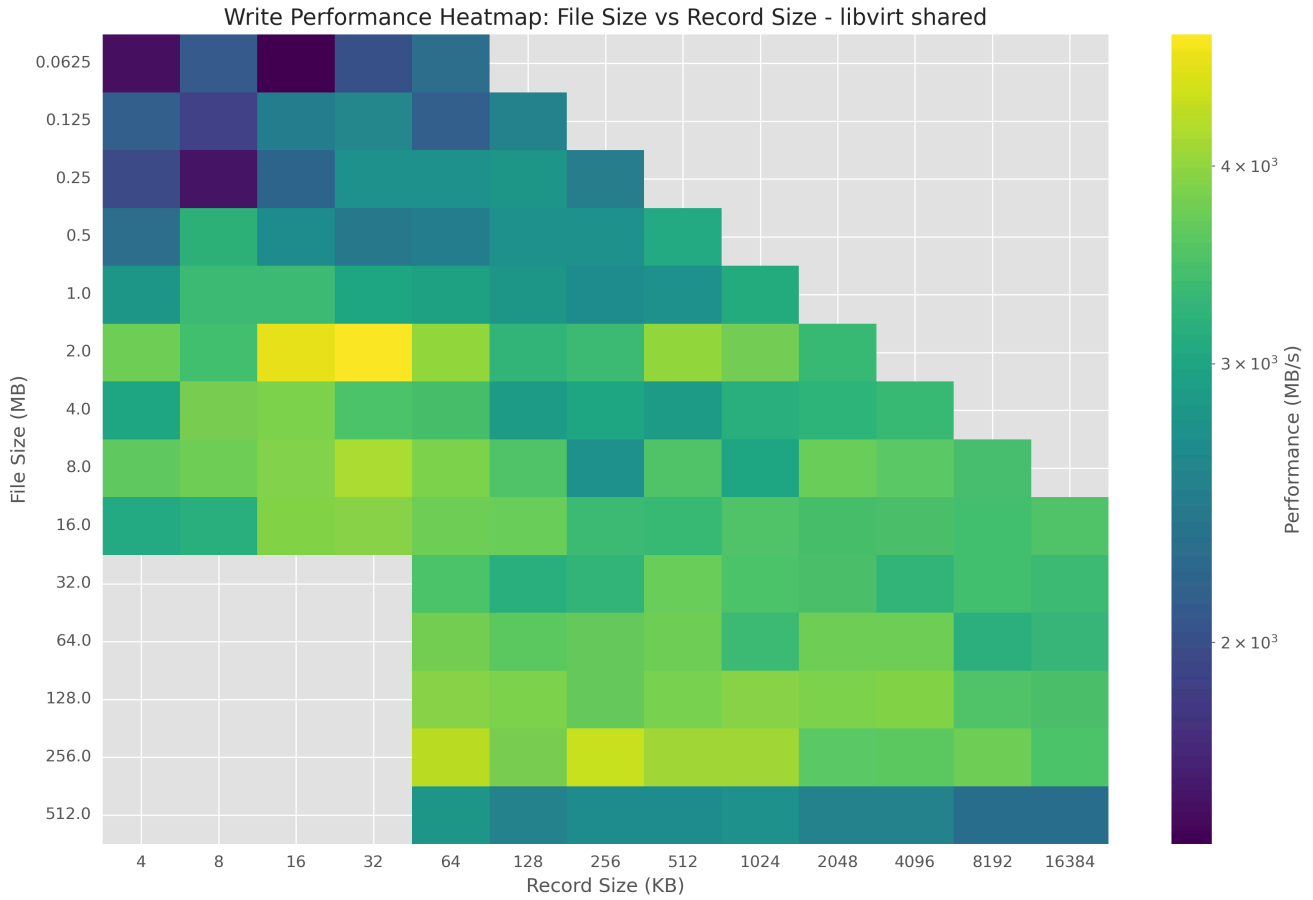



Read Performance Heatmap: File Size vs Record Size - libvirt local
File Size (MB)
0.0625
0.125
0.25
0.5
1.0
2.0
4.0
8.0
16.0
32.0
64.0
128.0
256.0
512.0
4
8
16
32
64
128
256
512
1024
2048
4096
8192
16384
Record Size (KB)
Performance (MB/s)
2×10^4
10^4
6×10^3


Write Performance Heatmap: File Size vs Record Size - libvirt local
File Size (MB)
0.0625
0.125
0.25
0.5
1.0
2.0
4.0
8.0
16.0
32.0
64.0
128.0
256.0
512.0
4
8
16
32
64
128
256
512
1024
2048
4096
8192
16384
Record Size (KB)
Performance (MB/s)
3×10^3
2×10^3


Performance vs File Size - libvirt shared
Performance (MB/s)
14000
12000
10000
8000
6000
4000
2000
0
0
2
8
32
128
512
File Size (MB)
read
write
reread
rewrite
rread


Read Performance Heatmap: File Size vs Record Size - libvirt shared
File Size (MB)
0.0625
0.125
0.25
0.5
1.0
2.0
4.0
8.0
16.0
32.0
64.0
128.0
256.0
512.0
4
8
16
32
64
128
256
512
1024
2048
4096
8192
16384
Record Size (KB)
Performance (MB/s)
10^4
10^3

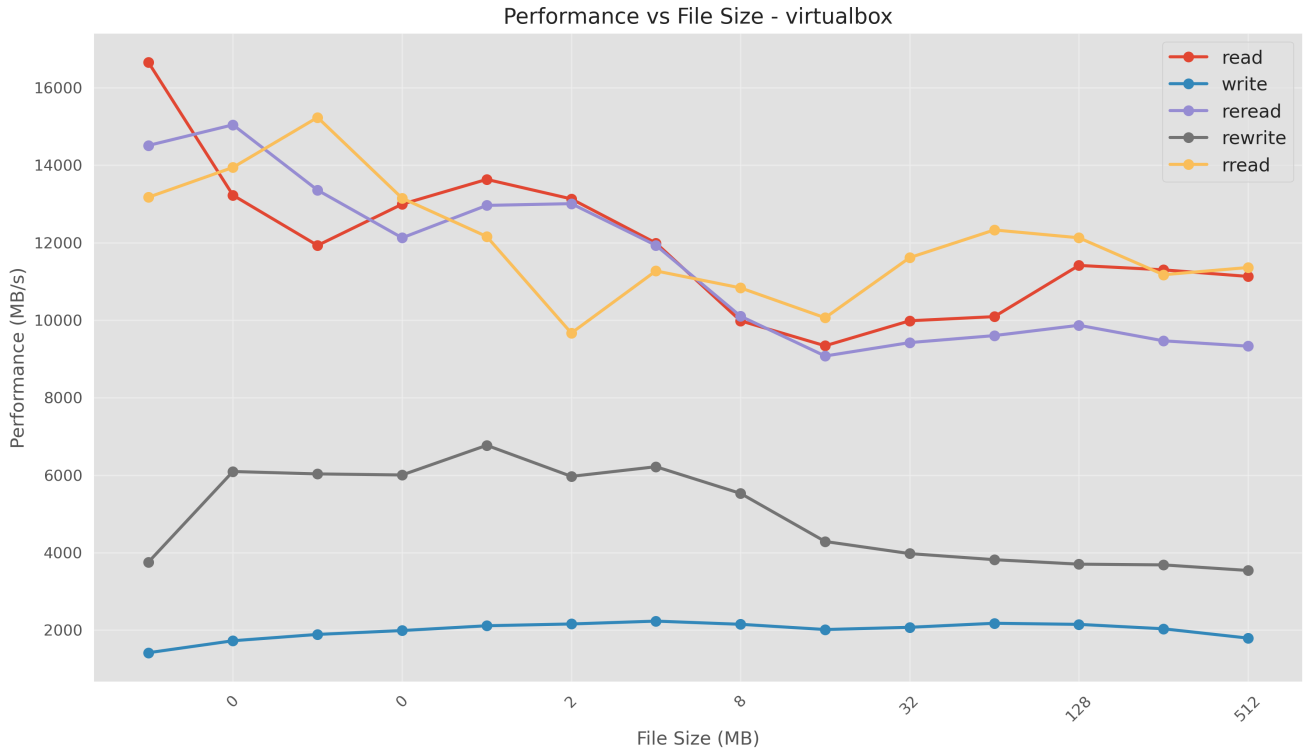


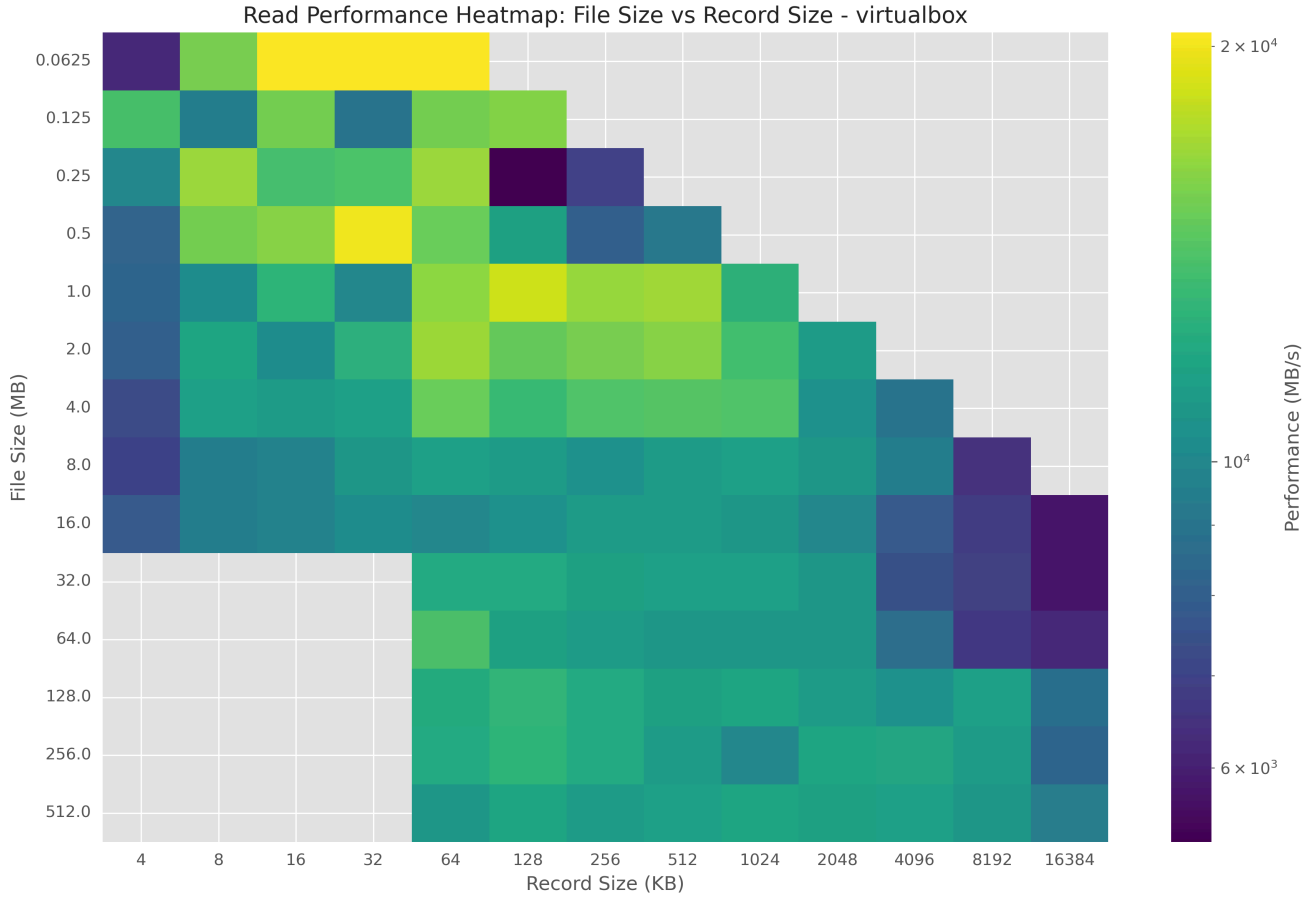
Virtualbox

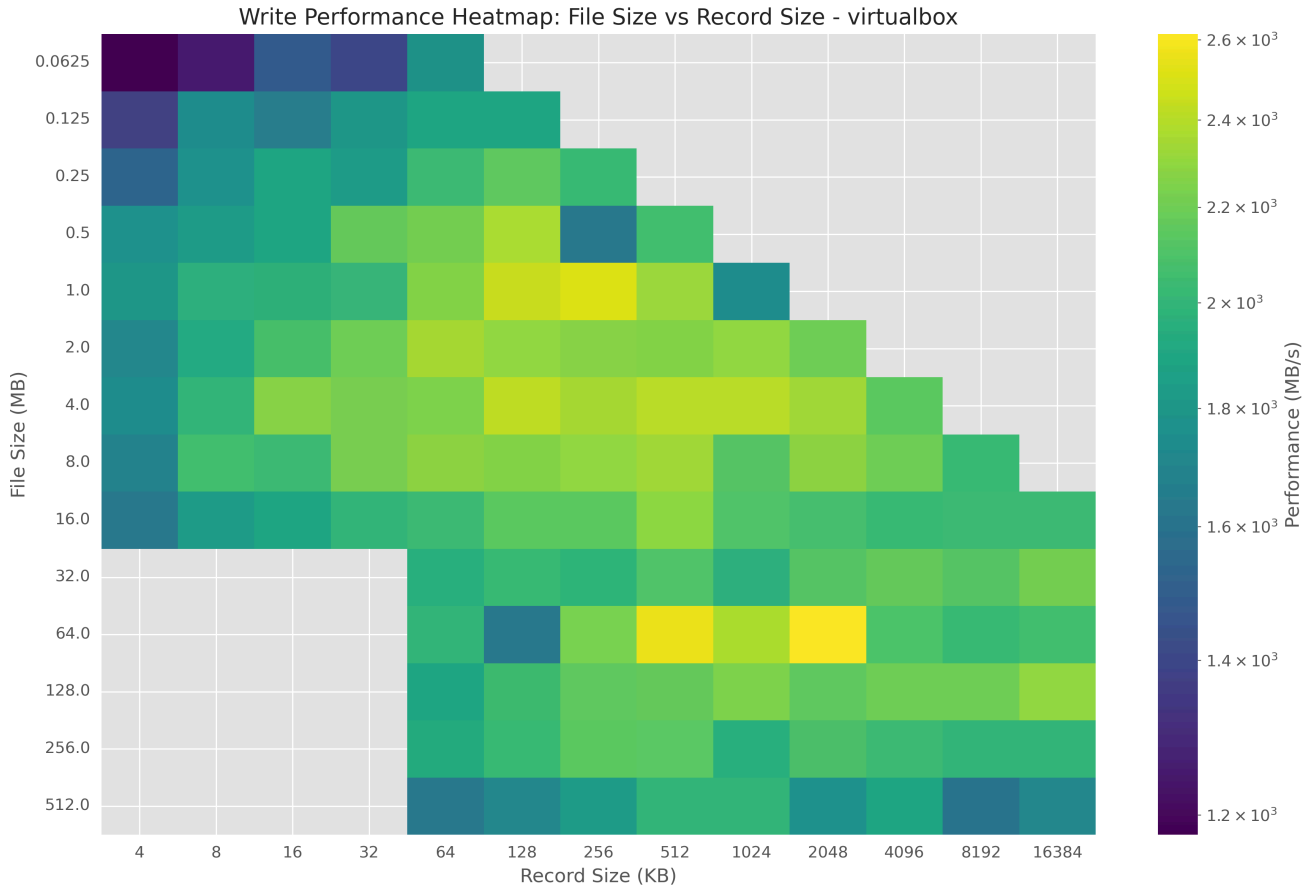
Performance Test Summary

Test Type	Specific Test	Metric	Value	Units
CPU	Standard CPU Stress Test	Bogo Ops/s (real time)	760.82	bogo ops/s
	Matrix Multiplication	Bogo Ops/s (real time)	441.64	bogo ops/s
	FFT CPU Test	Bogo Ops/s (real time)	4549.21	bogo ops/s
	Phi CPU Test	Bogo Ops/s (real time)	87463710.10	bogo ops/s
	Sysbench CPU Test	Events per second	4540.29	events/sec
Memory	VM Stress Test	Bogo Ops/s (real time)	71160.02	bogo ops/s
	Malloc Stress Test	Bogo Ops/s (real time)	33555.54	bogo ops/s
	Bigheap Stress Test	Bogo Ops/s (real time)	16093.94	bogo ops/s
	Sysbench Memory Test	Transfer Speed	50875.72	MiB/sec

Test Type	Specific Test	Metric	Value	Units
Network	iperf Bandwidth Test	Bandwidth	3.52	Gbits/sec
	ping Latency Test	RTT (avg)	0.343	ms
I/O	fio write Test	Write BW	94.0	MiB/s







Container

Project Structure

```
.
├── assignment.md
├── docker
│   ├── base-test
│   │   └── Dockerfile
│   ├── hpl-test
│   │   ├── Dockerfile
│   │   └── HPL.dat
│   └── iperf-server
│       └── Dockerfile
├── docker-compose.yaml
├── README.md
├── results
├── scripts
│   ├── run_test.sh
│   └── tests
│       ├── cpu_test.sh
│       ├── io_test.sh
│       ├── memory_test.sh
```

```
│       └── network_test.sh
└── setup.sh
```

Description

This project provides a self-contained, Dockerized environment for conducting a range of system performance benchmarks. It leverages `docker-compose` to define and orchestrate the necessary services (containers) and their network, offering a consistent and portable way to measure the performance characteristics of different environments or configurations without direct installation of benchmark tools on the host system.

The benchmarking environment, as defined by the `docker-compose.yaml` file, consists of the following services:

- **iperf-server:** This container is configured specifically to run the `iperf` network benchmarking tool in server mode. Its primary role is to act as the endpoint for network performance tests initiated from other containers, allowing for measurement of network bandwidth between services within the defined Docker network.
- **base-test:** This is the primary worker container for general system benchmarks. Its Dockerfile includes the installation of a comprehensive suite of benchmarking tools, including `stress-ng` (for CPU and memory stress), `sysbench` (for CPU and memory performance), `iozone3` and potentially `fio` (for disk I/O), and `iperf` and `iputils-ping` (for network testing). This container is where the core benchmark scripts (`cpu_test.sh`, `memory_test.sh`, `io_test.sh`, `network_test.sh`) are executed.
- **hpl-test:** This container is intended for running the High-Performance LINPACK (HPL) benchmark, a standard for rating the floating-point performance of computer systems. Its Dockerfile would include the necessary dependencies and the HPL benchmark itself. This service is designed for assessing heavy floating-point computation capabilities.

All containers operate within a dedicated `bm-network` (bridge driver) defined by Docker Compose, facilitating communication between services (e.g., the `base-test` container connecting to the `iperf-server`). A shared volume maps the `./results` directory on the host machine to `/benchmark/results` inside the containers, ensuring that all benchmark output is easily accessible and persisted outside the containers.

The provisioned environment is equipped to perform a suite of performance benchmarks using the scripts located in the `scripts/` directory (which are copied into the containers). These tests are designed to evaluate key performance aspects of the system the containers are running on:

- **CPU Performance:** These tests measure the processing capabilities using tools like `stress-ng` with various methods (standard stress, matrix multiplication, FFT, Phi) to apply different types of computational load, and `sysbench` for a general CPU performance evaluation.

- **Memory Performance:** This category assesses the memory subsystem's speed and capacity. `stress-ng` is used for various memory stress tests (VM, malloc, bigheap), and `sysbench` provides dedicated memory read/write tests.
- **I/O Performance:** Focusing on storage performance, these tests evaluate how efficiently the system can handle disk read and write operations. `iozone3` is used for filesystem-level testing, and `fio` provides capabilities for more granular block-level I/O testing. The performance observed here will reflect the performance of the underlying storage available to the Docker containers.
- **Network Performance:** Essential for understanding the connectivity, these tests measure network bandwidth using `iperf` between the `base-test` client and the `iperf-server`, and network latency using `ping`.
- **HPC Performance (HPL):** The `hpl-test` container is specifically set up to run the HPL benchmark, providing a measure of the system's high-performance floating-point computing capability.

Docker images

A Docker image is a lightweight, standalone, executable package that includes everything needed to run a piece of software, including the code, a runtime, libraries, environment variables, and configuration files. Think of it as a snapshot or a blueprint for a container.

Here are some key characteristics and concepts related to Docker images:

- **Read-Only Template**: An image is a read-only template. When you run a Docker image, you create a container, which is a running instance of the image. Any changes made while the container is running are stored in a writable layer on top of the image, leaving the original image unchanged.
- **Layered Structure**: Docker images are built up from a series of layers. Each instruction in a Dockerfile (the script used to build an image) creates a new layer. These layers are stacked on top of each other. This layered approach makes images efficient, as layers can be shared between different images, reducing disk space usage and speeding up builds.
- **Portability**: Docker images are designed to be portable. They encapsulate all dependencies, ensuring that an application will run consistently regardless of the environment where the image is deployed (as long as Docker is installed).
- **Built from Dockerfiles**: Docker images are typically built using a Dockerfile. A Dockerfile is a text file that contains a set of instructions on how to create the image. These instructions specify the base image, copy files, install dependencies, set environment variables, and define the command to run when the container starts.
- **Stored in Registries**: Docker images are stored in registries, such as Docker Hub (the default public registry) or private registries. These registries act as repositories where you can push and pull images.

base test

This Dockerfile creates a Docker image based on Ubuntu 20.04. It installs a suite of command-line benchmarking tools (`stress-ng` , `sysbench` , `iperf` , `iozone3` , `fio` , `iputils-ping`) along with `vim` . It sets up a `/benchmark` directory as the working directory, creates a `/scripts` directory, copies local scripts into `/benchmark/scripts` , makes those scripts executable, and configures the container to simply stay running by default using `sleep infinity` .

```
# Base image for benchmarks
FROM ubuntu:20.04
```

This instruction sets the base image for the subsequent instructions. It specifies that this new image will be built on top of the official Ubuntu 20.04 image available on Docker Hub.

```
# Set DEBIAN_FRONTEND to noninteractive to avoid prompts during package
installation
ARG DEBIAN_FRONTEND=noninteractive
```

This instruction defines a build-time variable `DEBIAN_FRONTEND` and sets its default value to noninteractive. This is a common practice in Dockerfiles when installing packages using `apt-get` to prevent the installation process from prompting the user for input, which would cause the build to hang.

```
# Install common tools needed for all performance tests
RUN apt-get update && apt-get install --no-install-recommends -y \
    vim \
    stress-ng \
    sysbench \
    iperf \
    iozone3 \
    fio \
    iputils-ping \
    && apt-get clean \
    && rm -rf /var/lib/apt/lists/* # Clean up apt cache
```

This is a multi-part instruction executed in a single `RUN` layer to minimize the number of layers and keep the image size down. Updates the list of available packages and their versions from the repositories and installs the specified packages

```
# Create the directory for results inside the container
RUN mkdir /benchmark
```

This instruction executes the `mkdir /benchmark` command inside the container, creating a directory named benchmark at the root filesystem level.

```
# Set the working directory
WORKDIR /benchmark
```

This instruction sets the working directory for any subsequent `RUN`, `CMD`, `ENTRYPOINT`, `COPY`, or `ADD` instructions that do not specify an absolute path. From this point onwards, commands will be executed relative to /benchmark inside the container.

```
# Create scripts directory
RUN mkdir -p /scripts
```

This instruction creates a directory named scripts at the root filesystem level. The `-p` flag ensures that parent directories are created if they don't exist.

```
# Copy benchmark scripts
COPY scripts ./scripts
```

This instruction copies files or directories from the build context (the directory on the host machine where the docker build command is run) into the Docker image.

```
# Make scripts executable
RUN chmod +x ./scripts/tests/*.sh
RUN chmod +x ./scripts/*.sh
```

This instructions makes all files ending with .sh executable.

```
# Default command to keep the container running
CMD ["sleep", "infinity"]
```

This instruction sets the default command that will be executed when a container is started from this image without specifying a command. ["sleep", "infinity"] is executed as an executable with arguments (exec form). This command simply tells the container to sleep indefinitely.

hpl-test

This Dockerfile creates a self-contained environment for running the HPL LINPACK benchmark. It starts from a clean Ubuntu 20.04 base, installs necessary build tools, then proceeds to download, build, and install optimized versions of OpenBLAS and OpenMPI from source. Finally, it downloads, configures, and compiles the HPL benchmark itself, linking against the custom-built libraries. The resulting image contains the HPL executable and is set up to allow easy execution of the benchmark.

```
# Base image: Ubuntu 20.04 LTS
FROM ubuntu:20.04
```

This instruction sets the base image for the subsequent instructions. It specifies that this new image will be built on top of the official Ubuntu 20.04 image available on Docker Hub.

```
# Prevent interactive prompts during package installation
ENV DEBIAN_FRONTEND=noninteractive
```

This instruction defines a build-time variable `DEBIAN_FRONTEND` and sets its default value to noninteractive. This is a common practice in Dockerfiles when installing packages using `apt-get` to prevent the installation process from prompting the user for input, which would cause the build to hang.

```
ENV OPENBLAS_VERSION=v0.3.26
ENV OPENMPI_VERSION=4.1.4
ENV HPL_VERSION=2.3
ENV OPT_DIR=/opt
ENV OPENBLAS_DIR=${OPT_DIR}/OpenBLAS
ENV MPI_HOME=${OPT_DIR}/OpenMPI
ENV HPL_DIR=${OPT_DIR}/hpl
ENV HPL_ARCH=linux
```

These instructions set environment variables within the Docker image. These variables define the versions of the software to be installed and specify the installation directories.

```
RUN apt-get update && apt-get install -y --no-install-recommends \
    build-essential \
    hwloc \
    libhwloc-dev \
    libevent-dev \
    gfortran \
    git \
    wget \
    tar \
    ca-certificates \
    openssh-client \
    vim \
    && rm -rf /var/lib/apt/lists/*
```

This instruction updates the package list and installs the necessary build dependencies and utility tools.

- `apt-get update`: Refreshes the list of available packages.

- `apt-get install -y --no-install-recommends ...` : Installs the specified packages without asking for confirmation (`-y`) and without installing recommended packages (`--no-install-recommends`). The packages include:
  - `build-essential` : Provides essential build tools like `gcc` , `g++` , and `make` .
  - `hwloc` , `libhwloc-dev` : Libraries for discovering hardware locality (useful for optimizing parallel applications).
  - `libevent-dev` : A library for asynchronous event notification.
  - `gfortran` : The GNU Fortran compiler, required for building HPL and potentially OpenMPI.
  - `git` : For cloning the OpenBLAS source code.
  - `wget` : For downloading source tarballs.
  - `tar` : For extracting tar archives.
  - `ca-certificates` : Essential for secure connections when downloading.
  - `openssh-client` : Included likely for potential future use in connecting to other nodes if running distributed HPL.
  - `vim` : A text editor, included for convenience in debugging or inspecting files within the container.
- `&& rm -rf /var/lib/apt/lists/*` : Cleans up the apt cache to reduce the final image size.

```
WORKDIR /tmp
RUN git clone https://github.com/xianyi/OpenBLAS.git && \
    cd OpenBLAS && \
    git checkout ${OPENBLAS_VERSION} && \
    # Use all available CPU cores for faster build
    make -j$(nproc) && \
    make PREFIX=${OPENBLAS_DIR} install && \
    cd / && \
    # Clean up source directory to reduce image size
    rm -rf /tmp/OpenBLAS
```

`WORKDIR /tmp` : This instruction sets the working directory to /tmp for the following instructions

- `git clone https://github.com/xianyi/OpenBLAS.git` : Clones the OpenBLAS source code repository from GitHub into `/tmp/OpenBLAS` .
- `cd OpenBLAS` : Changes the current directory to the cloned OpenBLAS source directory.
- `git checkout ${OPENBLAS_VERSION}` : Switches to the specific version of OpenBLAS defined by the `OPENBLAS_VERSION` environment variable.
- `make -j$(nproc)` : Compiles OpenBLAS using `make` . The `-j$(nproc)` option tells `make` to use as many parallel jobs as there are CPU cores available in the build environment, speeding up compilation.

- `make PREFIX=${OPENBLAS_DIR} install` : Installs the compiled OpenBLAS libraries and headers to the path specified by the `OPENBLAS_DIR` environment variable (`/opt/OpenBLAS`).
- `cd /` : Changes the directory back to the root filesystem.
- `rm -rf /tmp/OpenBLAS` : Removes the OpenBLAS source directory from `/tmp` to keep the image size down.

```
WORKDIR /tmp
RUN wget https://download.open-mpi.org/release/open-
mpi/v${OPENMPI_VERSION%.*}/openmpi-${OPENMPI_VERSION}.tar.gz && \
    tar xzf openmpi-${OPENMPI_VERSION}.tar.gz && \
    cd openmpi-${OPENMPI_VERSION} && \
    # Enable aggressive optimization
    CFLAGS="-Ofast" ./configure --prefix=${MPI_HOME} --enable-mpi-cxx && \
    make -j$(nproc) all && \
    make install && \
    cd / && \
    # Clean up source files to reduce image size
    rm -rf /tmp/openmpi-${OPENMPI_VERSION}.tar.gz
/tmp/openmpi-${OPENMPI_VERSION}
```

`WORKDIR /tmp` : Sets the working directory back to `/tmp` for the OpenMPI build process.

- `wget https://download.open-mpi.org/release/open-mpi/v${OPENMPI_VERSION%.*}/openmpi-${OPENMPI_VERSION}.tar.gz` : Downloads the OpenMPI source tarball from the specified URL. The `${OPENMPI_VERSION%.*}` syntax extracts the major and minor version from the `OPENMPI_VERSION` variable (e.g., `4.1` from `4.1.4`).
- `tar xzf openmpi-${OPENMPI_VERSION}.tar.gz` : Extracts the downloaded tarball.
- `cd openmpi-${OPENMPI_VERSION}` : Changes the directory to the extracted OpenMPI source.
- `CFLAGS="-Ofast" ./configure --prefix=${MPI_HOME} --enable-mpi-cxx` : Configures OpenMPI for the build.
  - `CFLAGS="-Ofast"` : Sets the C compiler flags for aggressive optimization.
  - `--prefix=${MPI_HOME}` : Specifies the installation directory using the `MPI_HOME` environment variable (`/opt/OpenMPI`).
  - `--enable-mpi-cxx` : Enables the C++ bindings for MPI.
- `make -j$(nproc) all` : Compiles OpenMPI using all available CPU cores.
- `make install` : Installs the compiled OpenMPI libraries and binaries to the specified prefix.
- `cd /` : Changes the directory back to the root filesystem.
- `rm -rf /tmp/openmpi-${OPENMPI_VERSION}.tar.gz /tmp/openmpi-${OPENMPI_VERSION}` : Removes the downloaded tarball and source

directory to reduce image size.

```
# Add OpenMPI binaries and libraries to PATH and LD_LIBRARY_PATH
ENV PATH=${PATH:-}:${MPI_HOME}/bin
ENV LD_LIBRARY_PATH=${MPI_HOME}/lib
```

`ENV PATH=PATH:-:{MPI_HOME}/bin` : This instruction updates the PATH environment variable. It prepends the directory `${MPI_HOME}/bin` (where OpenMPI binaries are installed) to the existing `PATH`. The `${PATH:-}` syntax ensures that if `PATH` is not already set, it defaults to an empty string before prepending the new directory. This allows executables like mpicc and mpirun to be found without specifying their full path.

`ENV LD_LIBRARY_PATH=${MPI_HOME}/lib` : This instruction sets the `LD_LIBRARY_PATH` environment variable. It adds the directory `${MPI_HOME}/lib` (where OpenMPI libraries are installed) to this variable. This helps the system find the necessary shared libraries at runtime.

```
WORKDIR ${OPT_DIR}
RUN wget https://netlib.org/benchmark/hpl/hpl-${HPL_VERSION}.tar.gz && \
    tar xzf hpl-${HPL_VERSION}.tar.gz && \
    mv hpl-${HPL_VERSION} ${HPL_DIR} && \
    rm hpl-${HPL_VERSION}.tar.gz && \
    cd ${HPL_DIR}/setup && \
    sh make_generic && \
    cp Make.UNKNOWN ../Make.${HPL_ARCH} && \
    cd .. && \
    # Configure HPL Makefile for our environment
    sed -i "s|^ARCH\s* =.*|ARCH = ${HPL_ARCH}|" Make.${HPL_ARCH} && \
    sed -i "s|^TOPdir\s* =.*|TOPdir = ${HPL_DIR}|" Make.${HPL_ARCH} && \
    # Configure MPI settings
    sed -i "s|^MPdir\s* =.*|MPdir = ${MPI_HOME}|" Make.${HPL_ARCH} && \
    sed -i "s|^MPinc\s* =.*|MPinc = -I\$(MPdir)/include|" Make.${HPL_ARCH}
&& \
    sed -i "s|^MPlib\s* =.*|MPlib = \$(MPdir)/lib/libmpi.so|"
Make.${HPL_ARCH} && \
    # Configure Linear Algebra library (OpenBLAS) settings
    sed -i "s|^LAdir\s* =.*|LAdir = ${OPENBLAS_DIR}|" Make.${HPL_ARCH} &&
\
    sed -i "s|^LAinc\s* =.*|LAinc =|" Make.${HPL_ARCH} && \
    # Use static OpenBLAS library for better performance
    sed -i "s|^LAlib\s* =.*|LAlib = \$(LAdir)/lib/libopenblas.a|"
Make.${HPL_ARCH} && \
    # Build HPL with all available cores
    make arch=${HPL_ARCH} -j$(nproc)
```

`WORKDIR ${OPT_DIR}` : Sets the working directory to /opt for the HPL build process.

- `wget https://netlib.org/benchmark/hpl/hpl-${HPL_VERSION}.tar.gz` : Downloads the HPL source tarball.
- `tar xzf hpl-${HPL_VERSION}.tar.gz` : Extracts the tarball.
- `mv hpl-${HPL_VERSION} ${HPL_DIR}` : Renames the extracted directory to the path specified by `HPL_DIR` (`/opt/hpl`).
- `rm hpl-${HPL_VERSION}.tar.gz` : Removes the downloaded tarball.
- `cd ${HPL_DIR}/setup` : Changes directory to the HPL setup directory.
- `sh make_generic` : Runs a script to create a generic Makefile template (`Make.UNKNOWN`).
- `cp Make.UNKNOWN ../Make.${HPL_ARCH}` : Copies the generic Makefile template to the main HPL directory and renames it based on the `HPL_ARCH` variable (`Make.linux`).
- `cd ..` : Changes directory back to the main HPL directory.
- `sed -i` : Configure HPL Makefile for our environment
- `make arch=${HPL_ARCH} -j$(nproc)` : Compiles HPL using the configured `Make.linux` file and all available CPU cores.

```
# Set working directory to HPL binary location
WORKDIR ${HPL_DIR}/bin/${HPL_ARCH}
```

This instruction sets the default working directory for the container to where the compiled HPL binary `xhpl` will be located `/opt/hpl/bin/linux` . This makes it easy to run the benchmark executable directly after starting the container.

```
# Copy the HPL configuration file with proper ownership
COPY ./docker/hpl-test/HPL.dat .
```

Copy an HPL.dat configuration file from the specified path in the build context into the current working directory `/opt/hpl/bin/linux` inside the container. This file contains the parameters for running the HPL benchmark.

```
CMD ["sleep", "infinity"]
```

This instruction sets the default command that will be executed when a container is started from this image without specifying a command. ["sleep", "infinity"] is executed as an executable with arguments (exec form). This command simply tells the container to sleep indefinitely.

iperf-server

This Dockerfile creates a minimal Docker image based on Ubuntu 20.04. It installs only the `iperf` package, sets up a `/benchmark` directory, and configures the container to

automatically start `iperf` in server mode when run. This image is designed to be a simple, dedicated iperf server for network benchmarking.

```
# Base image: Ubuntu 20.04 LTS
FROM ubuntu:20.04
```

This instruction sets the base image for the subsequent instructions. It specifies that this new image will be built on top of the official Ubuntu 20.04 image available on Docker Hub.

```
# Prevent interactive prompts during package installation
ENV DEBIAN_FRONTEND=noninteractive
```

This instruction defines a build-time variable `DEBIAN_FRONTEND` and sets its default value to noninteractive. This is a common practice in Dockerfiles when installing packages using `apt-get` to prevent the installation process from prompting the user for input, which would cause the build to hang.

```
# Install common tools needed for all performance tests
# Using --no-install-recommends reduces image size
RUN apt-get update && apt-get install --no-install-recommends -y \
    iperf \
    && apt-get clean \
    && rm -rf /var/lib/apt/lists/* # Clean up apt cache
```

This instruction updates the package list and installs the `iperf` package.

```
# Create the directory for results inside the container
RUN mkdir /benchmark
```

This instruction executes the `mkdir /benchmark` command inside the container, creating a directory named benchmark at the root filesystem level. This directory is likely intended to store benchmark results, though the subsequent `WORKDIR` changes.

```
# Set the working directory
WORKDIR /benchmark
```

This instruction sets the working directory for any subsequent `RUN`, `CMD`, `ENTRYPOINT`, `COPY`, or `ADD` instructions that do not specify an absolute path. From this point onwards, commands will be executed relative to /benchmark inside the container.

```
CMD ["iperf", "--server"]
```

This instruction sets the default command that will be executed when a container is started from this image without specifying a command. `["iperf", "--server"]` is executed as an executable with arguments (exec form). This command starts the `iperf` program in server mode, listening for incoming connections for network performance testing.

Docker compose

A Docker Compose file is a YAML file used to define and manage multi-container Docker applications. In a Compose file, you define:

- **Services**: Each service represents a container for a part of your application (e.g., a web server, a database, a cache). You specify the Docker image to use, ports to expose, volumes to mount, dependencies on other services, and more.
- **Networks**: You can define custom networks to allow containers to communicate with each other in an isolated environment.
- **Volumes**: You define volumes for persisting data generated by your containers, ensuring data isn't lost when containers are stopped or removed.

```yaml
services:
  iperf-server:
    image: iperf-server
    container_name: iperf-server
    volumes:
      - ./results:/benchmark/results
    deploy:
      resources:
        limits:
          cpus: '2'
          memory: 2G
    networks:
      - bm-network

  base-test:
    image: base-test
    container_name: base-test
    volumes:
      - ./results:/benchmark/results
    deploy:
      resources:
        limits:
          cpus: '4'
          memory: 4G
    networks:
      - bm-network

  hpl-test:
    image: hpl-test
```

```
    container_name: hpl-test
    volumes:
      - ./results:/benchmark/results
    deploy:
      resources:
        limits:
          cpus: '4'
          memory: 4G
    networks:
      - bm-network

networks:
  bm-network:
    driver: bridge
```

- `services:`
  - `image: base-test` : This specifies the Docker image to use for this service. When Docker Compose runs this configuration, it will look for an image named `base-test` locally.
  - `container_name: base-test` : This assigns a specific, static name to the container that will be created from this service.
  - `volumes:` : This section configures volume mounts for the container.
    - `- ./results:/benchmark/results` : This defines a bind mount. It means the directory `./results` on the host machine (relative to the location of your Docker Compose file) will be mounted inside the container at the path `/benchmark/results`.
  - `deploy:` : This section specifies deployment configurations for the service.
    - `resources:` : Configures resource constraints for the service.
      - `limits:` : Sets hard limits on the resources the container can use. Docker will prevent the container from exceeding these limits.
        - `cpus: '4'` : Limits the container to using a maximum of 4 CPU cores.
        - `memory: 4G` : Limits the container to using a maximum of 4 gigabytes of RAM.
  - `networks:` : This section specifies which networks the service's containers should connect to.
    - `- bm-network` : This indicates that the `base-test` container will be attached to a network named `bm-network`.
    -
- `networks:` : This is the section for defining or referencing networks.
  - `bm-network:` : This is the specific network being defined and named.
    - `driver: bridge` : This specifies that it's a standard bridge network. This creates a private, internal network on the Docker host that allows containers

connected to it to communicate with each other easily using their service names.

Managing the containers

Building docker images:

```
> ./setup.sh
```

Run the containers

```
> docker compose up -d
```

Stop the containers

```
> docker compose down
```

Remove images

```
> docker rmi iperf-server hpl-test base-test
```

Benchmarking

Run the test suit

```
> docker exec -it hpl-test bash
root@<containerid>:/opt/hpl/bin/linux> mpirun --allow-run-as-root -np 4
./xhpl 2>&1 | tee -a /benchmark/results/hpl-test.log
root@<containerid>:/opt/hpl/bin/linux> logout
```

Run hpl test

```
> docker exec -it base-test bash
root@9ea4555ef292:/benchmark> cd scripts/
root@9ea4555ef292:/benchmark> ./run_tests.sh
root@9ea4555ef292:/benchmark> logout
```

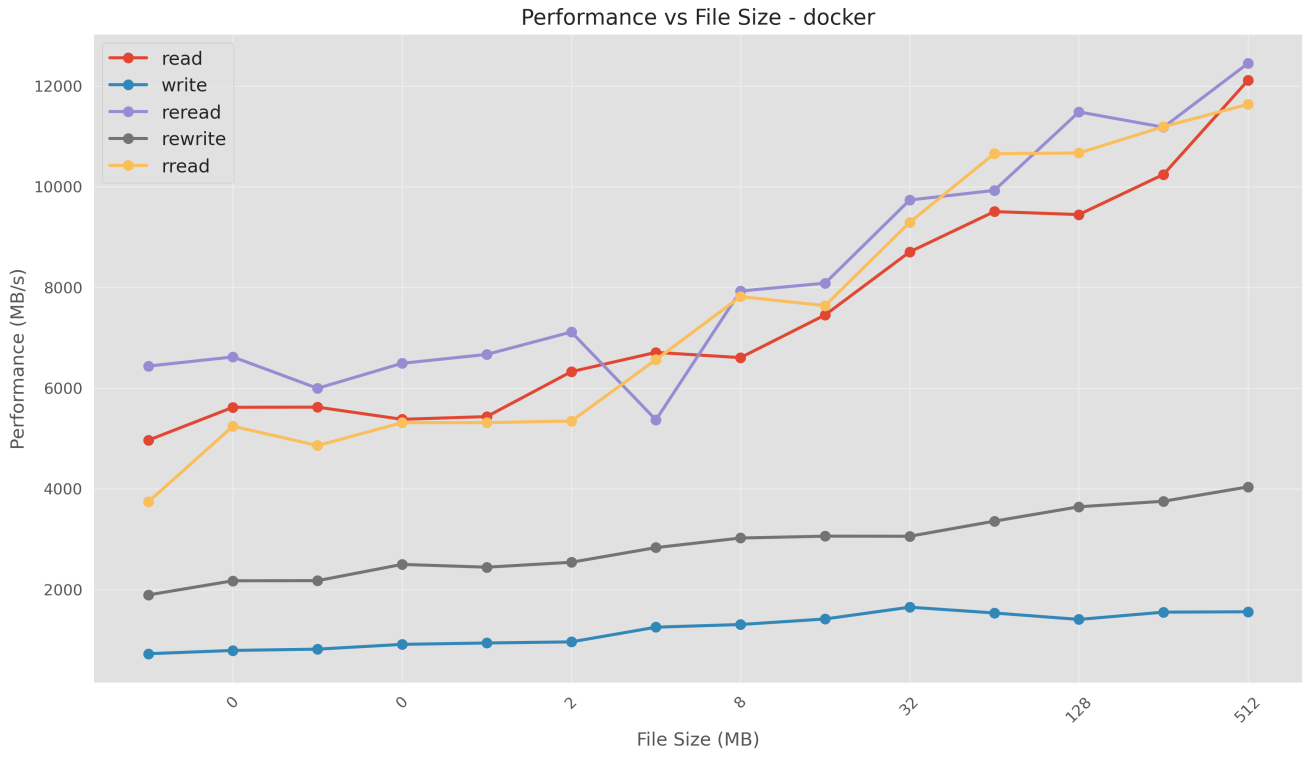
Retrieve the results

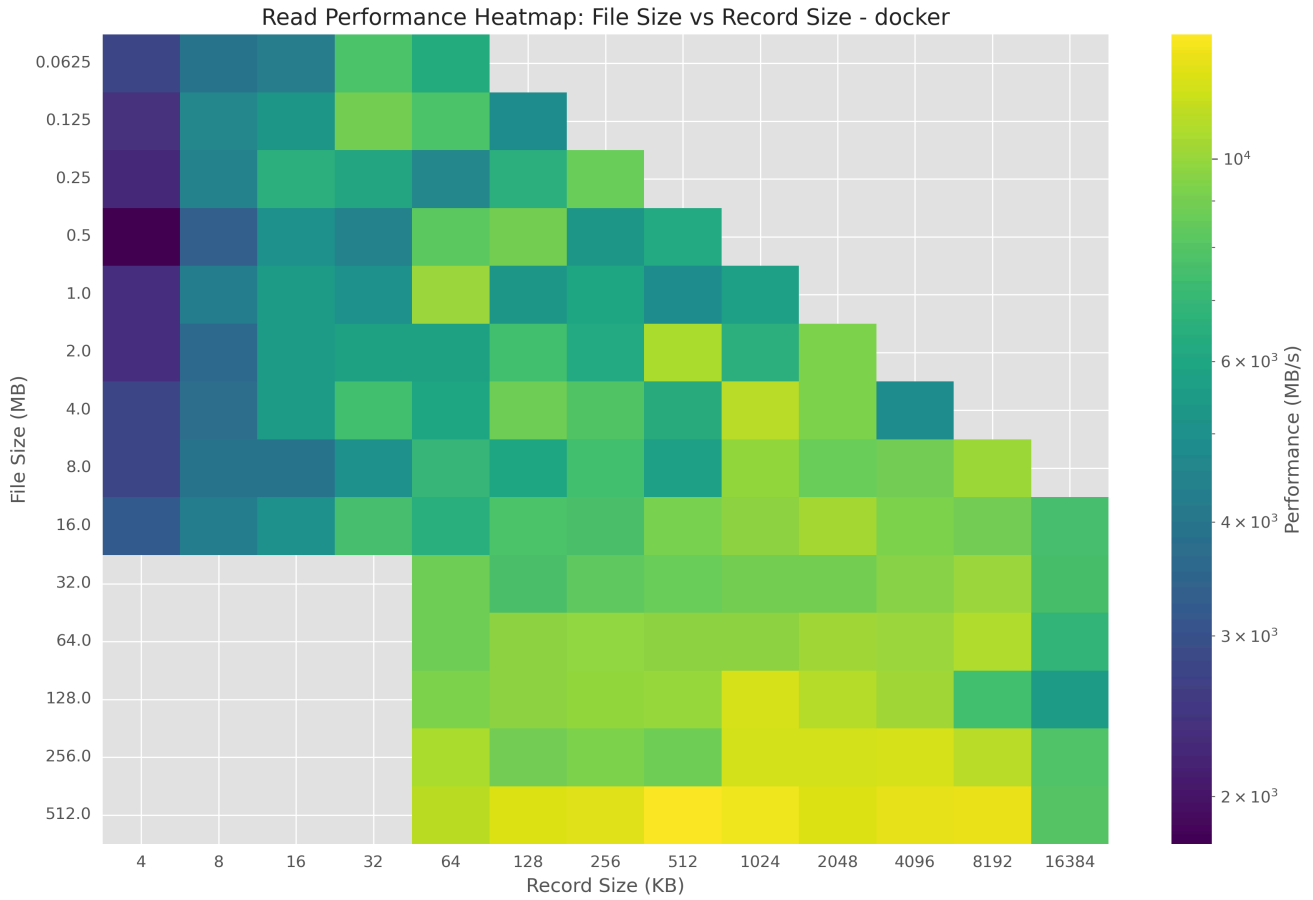
The results are stored in `results`

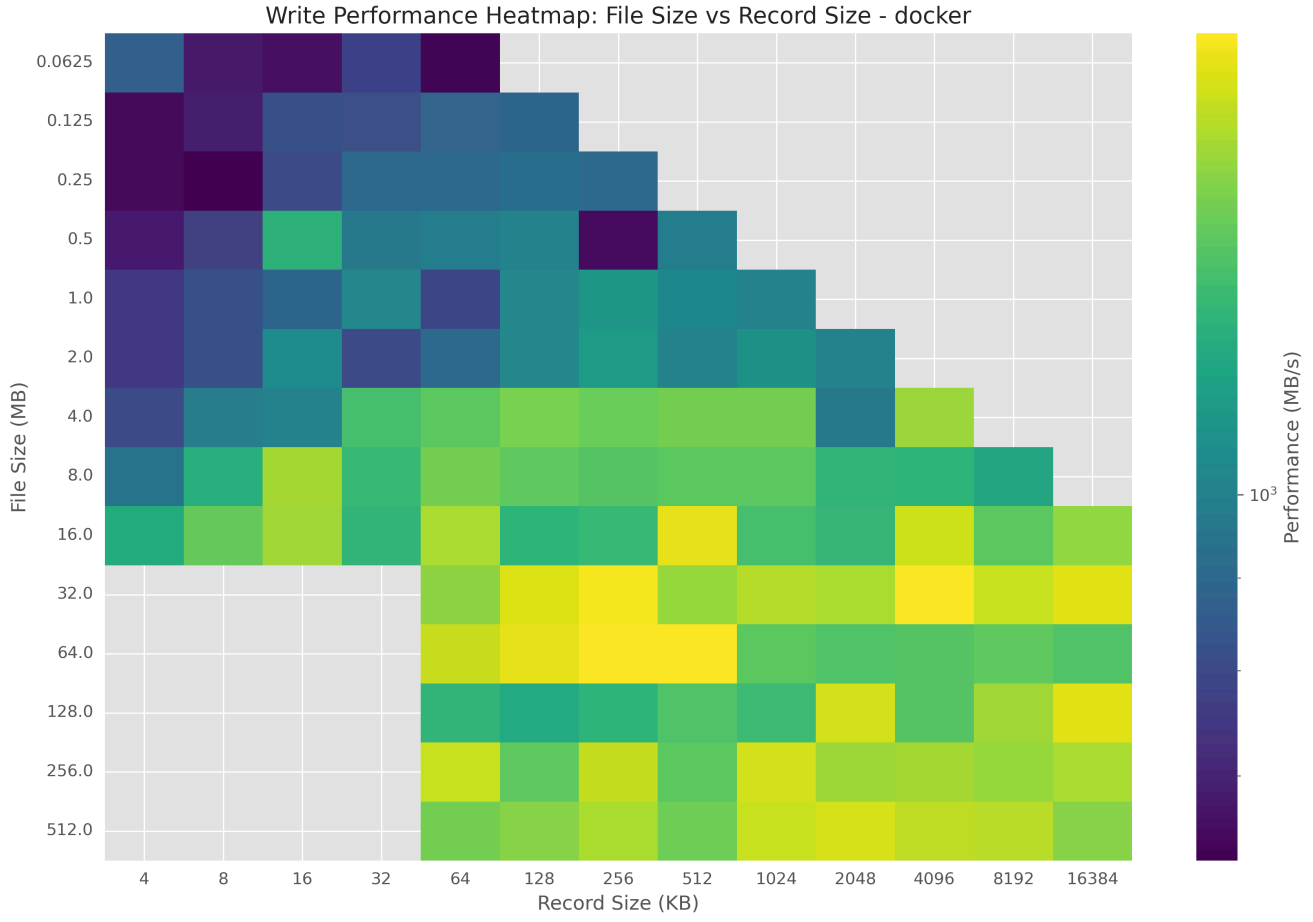
Results

Performance Test Summary

Test Type	Specific Test	Metric	Value	Units
CPU	Standard CPU Stress Test	Bogo Ops/s (real time)	1310.85	bogo ops/s
	Matrix Multiplication	Bogo Ops/s (real time)	484.22	bogo ops/s
	FFT CPU Test	Bogo Ops/s (real time)	5030.55	bogo ops/s
	Phi CPU Test	Bogo Ops/s (real time)	96900008.20	bogo ops/s
	Sysbench CPU Test	Events per second	4978.84	events/sec
Memory	VM Stress Test	Bogo Ops/s (real time)	71141.38	bogo ops/s
	Malloc Stress Test	Bogo Ops/s (real time)	42760.82	bogo ops/s
	Bigheap Stress Test	Bogo Ops/s (real time)	6876.28	bogo ops/s
	Sysbench Memory Test	Transfer Speed	60014.39	MiB/sec
Network	iperf Bandwidth Test	Bandwidth	27.6	Gbits/sec
	ping Latency Test	RTT (avg)	0.115	ms
I/O	fio write Test	Write BW	338	MiB/s



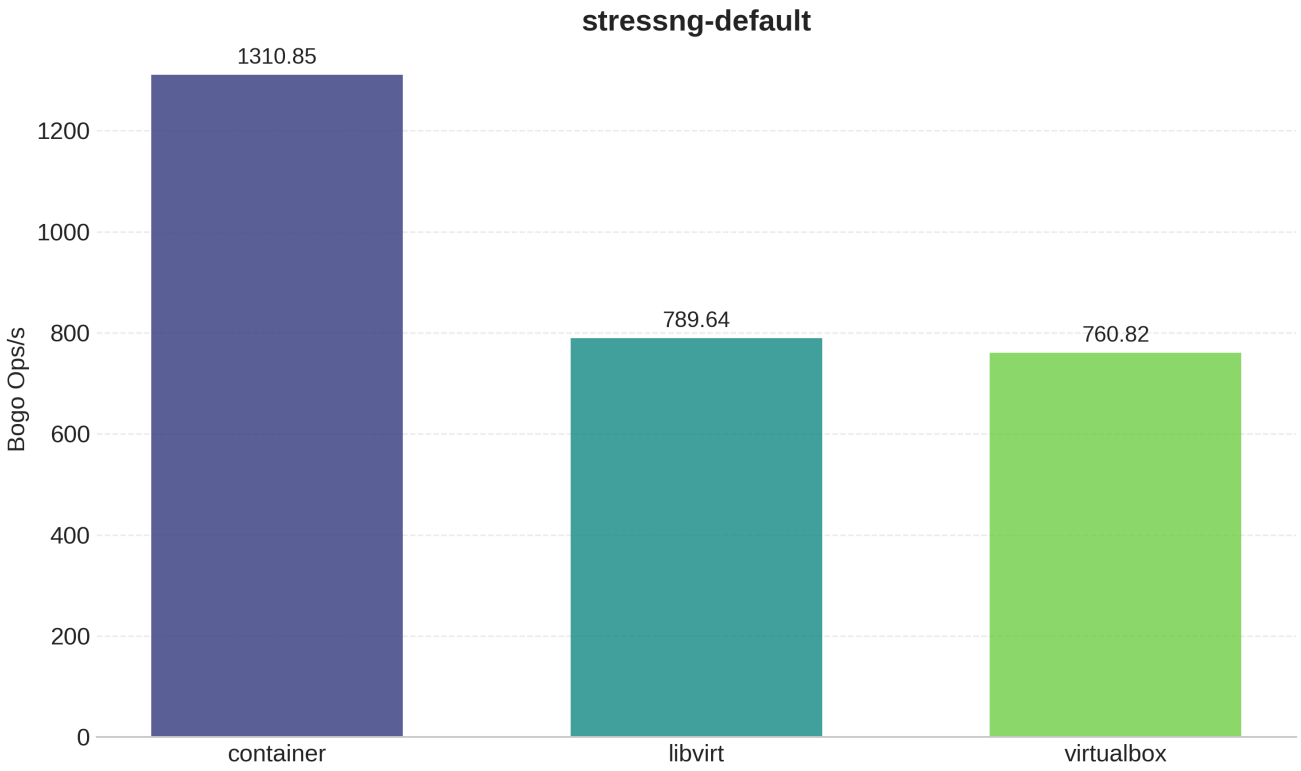




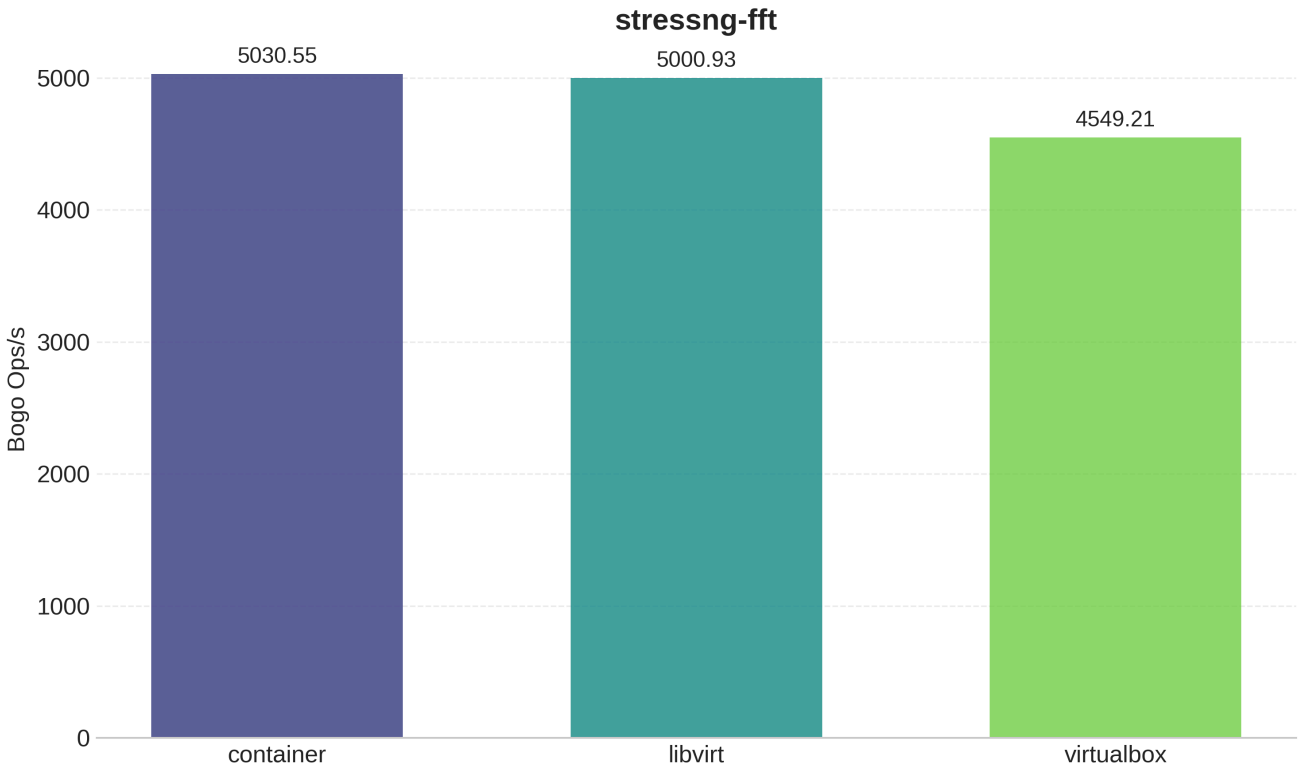
Performance Comparison

CPU

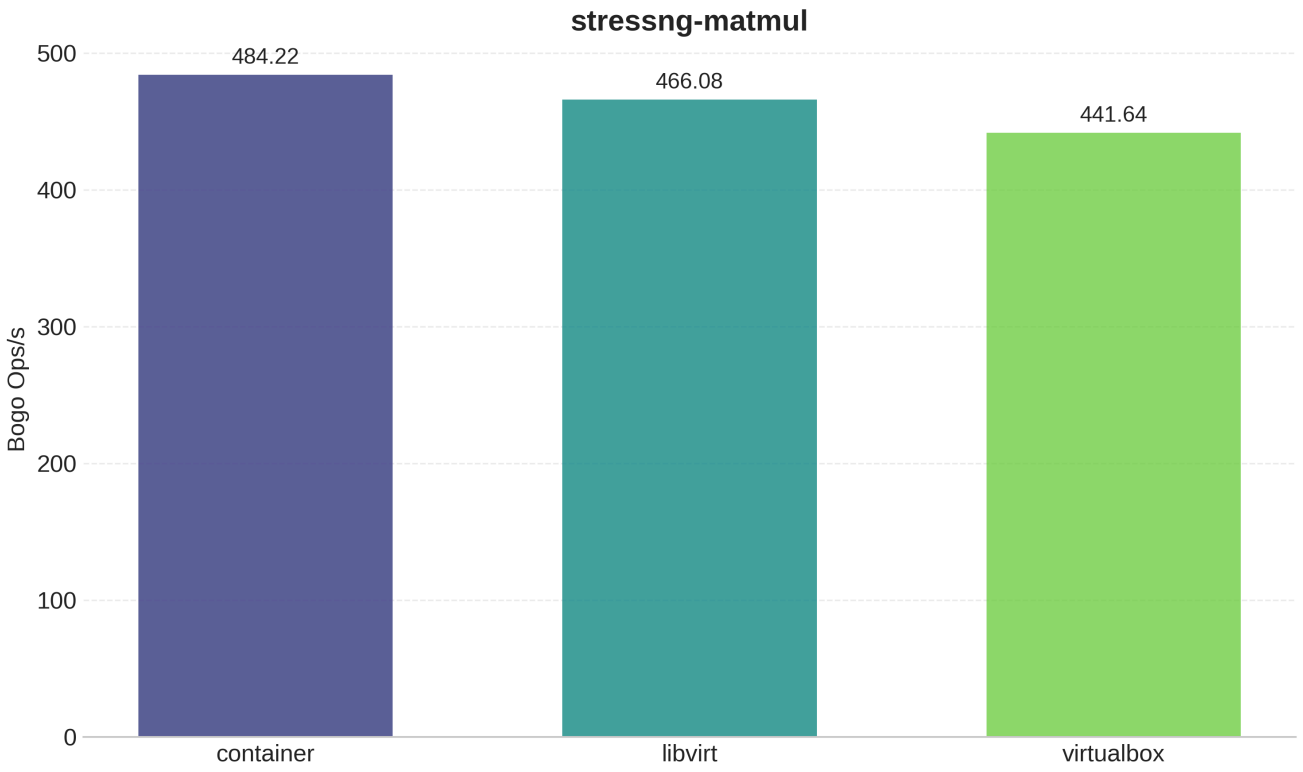
```
stress-ng --cpu 4 --timeout 60s
```



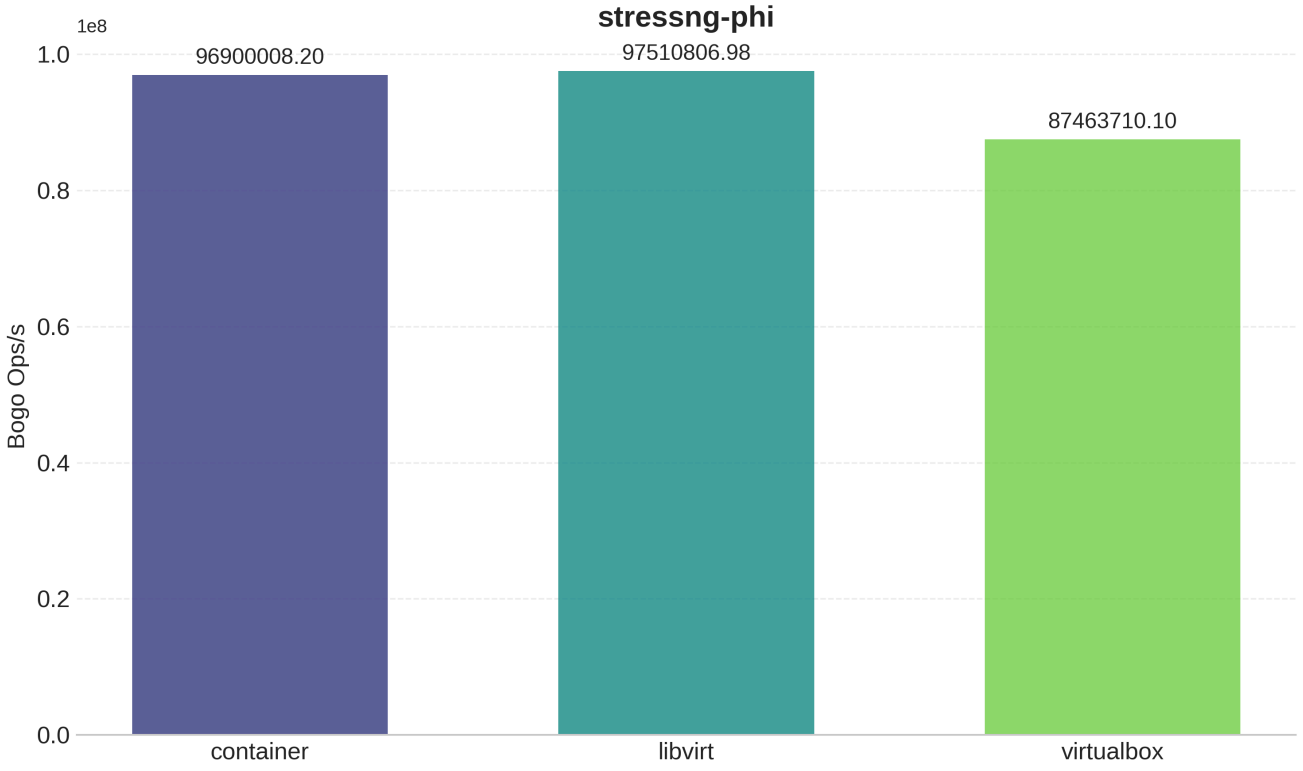

```
stress-ng --cpu 4 --cpu-method fft --timeout 60s
```



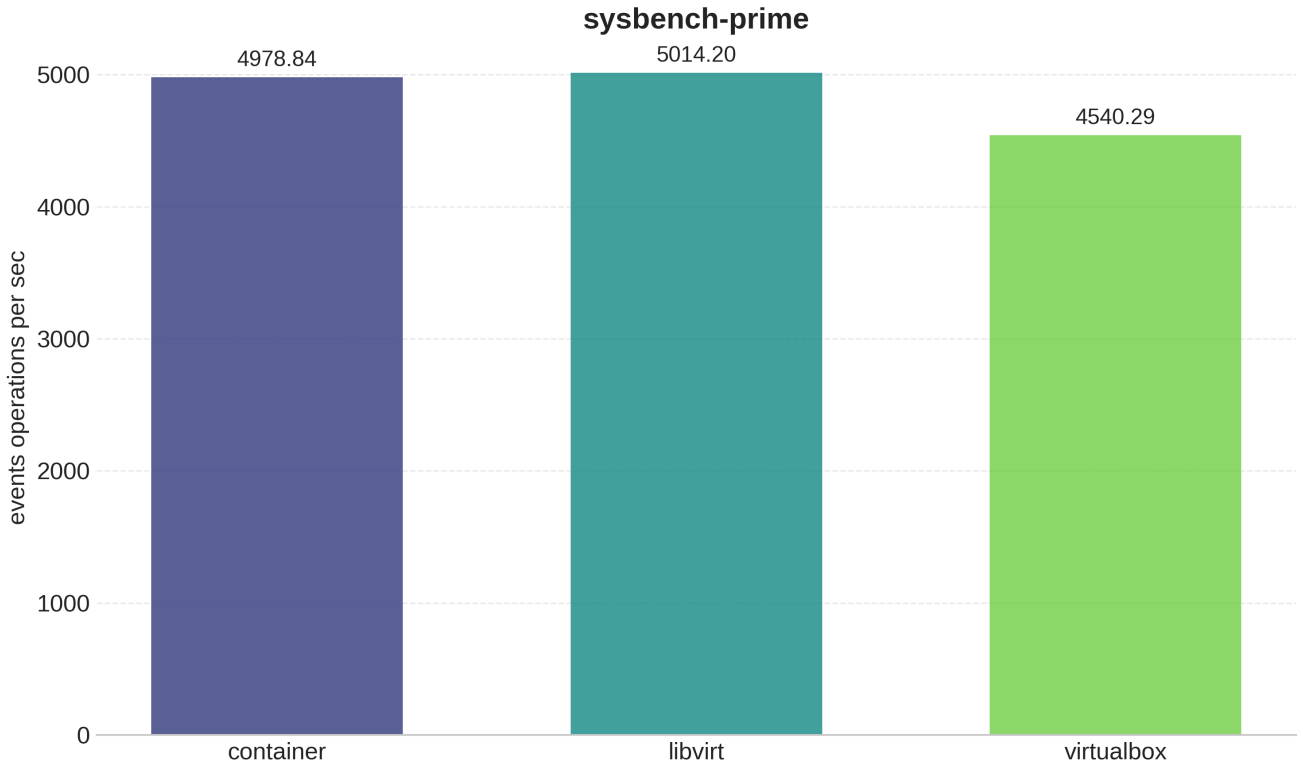

```
stress-ng --cpu 4 --cpu-method matrixprod --timeout 60s
```




```
stress-ng --cpu 4 --cpu-method phi --timeout 60s
```

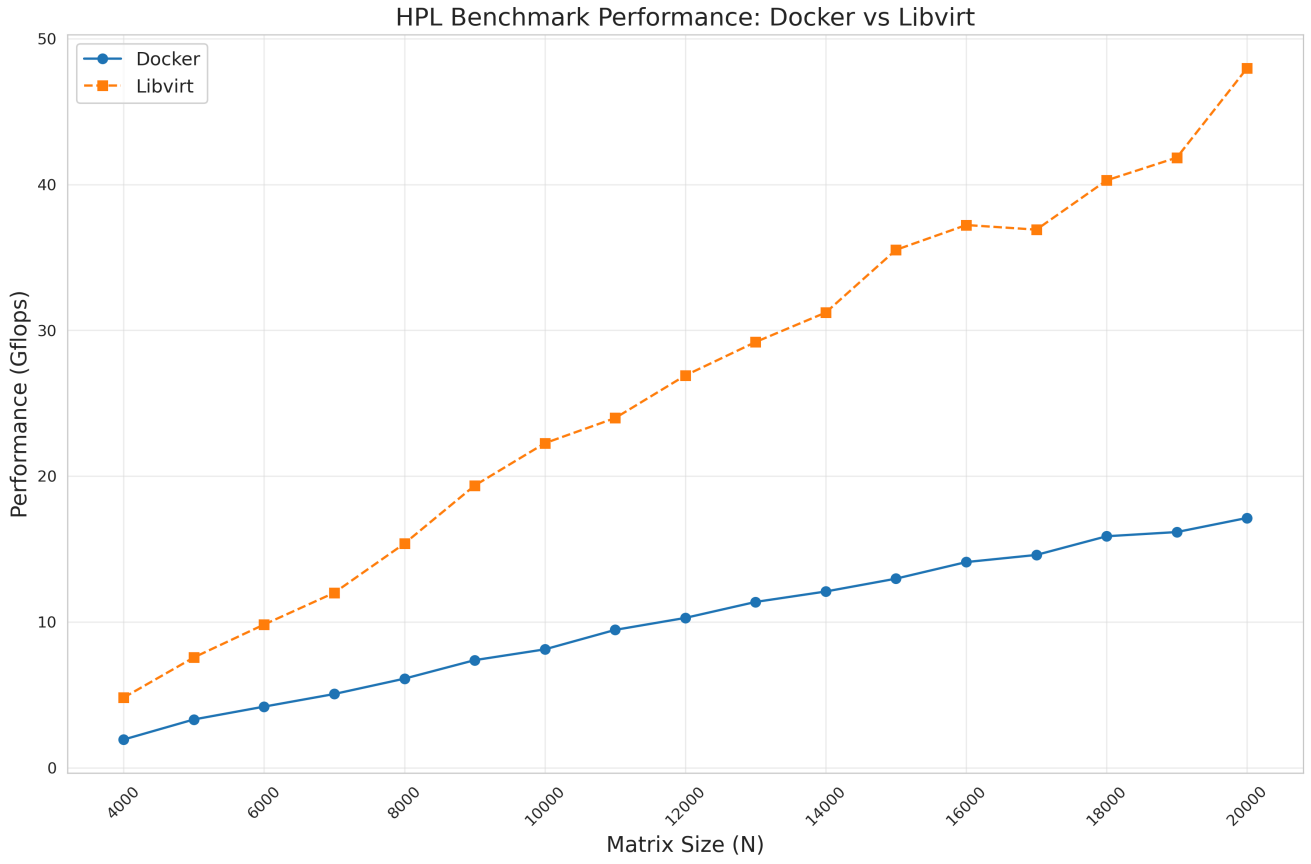



```
sysbench cpu --cpu-max-prime=20000 --threads=4 --time=60
```

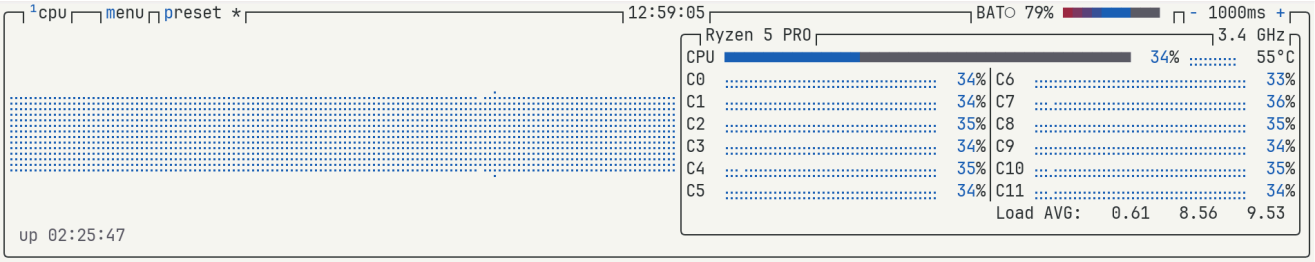


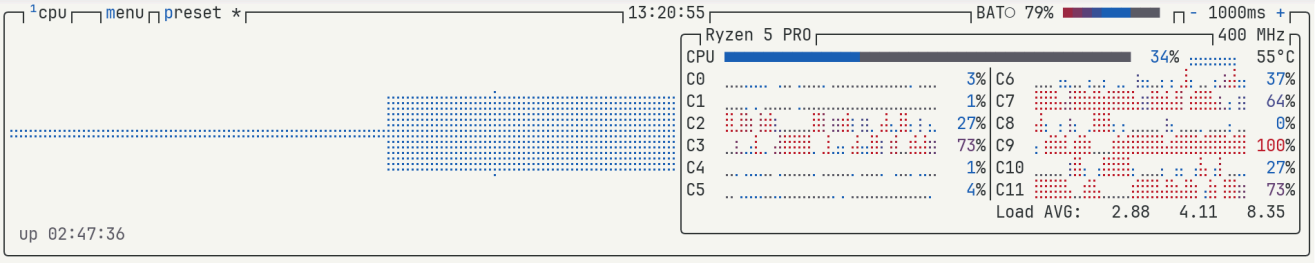
- The **container** environment generally shows strong performance, particularly excelling in the `stressng-default` test where it significantly outperformed both virtual machines.
- The **libvirt** virtual machine is highly competitive with the container, performing similarly or slightly better in tests like `stressng-phi` and `sysbench-prime`, and notably better than VirtualBox in tests like `stressng-fft`.
- The **VirtualBox** virtual machine consistently exhibited the lowest performance across all the benchmark tests shown (stressng-default, stressng-fft, stressng-matmul, stressng-phi, and sysbench-prime).

```
mpirun -np 4 ./xhpl
```



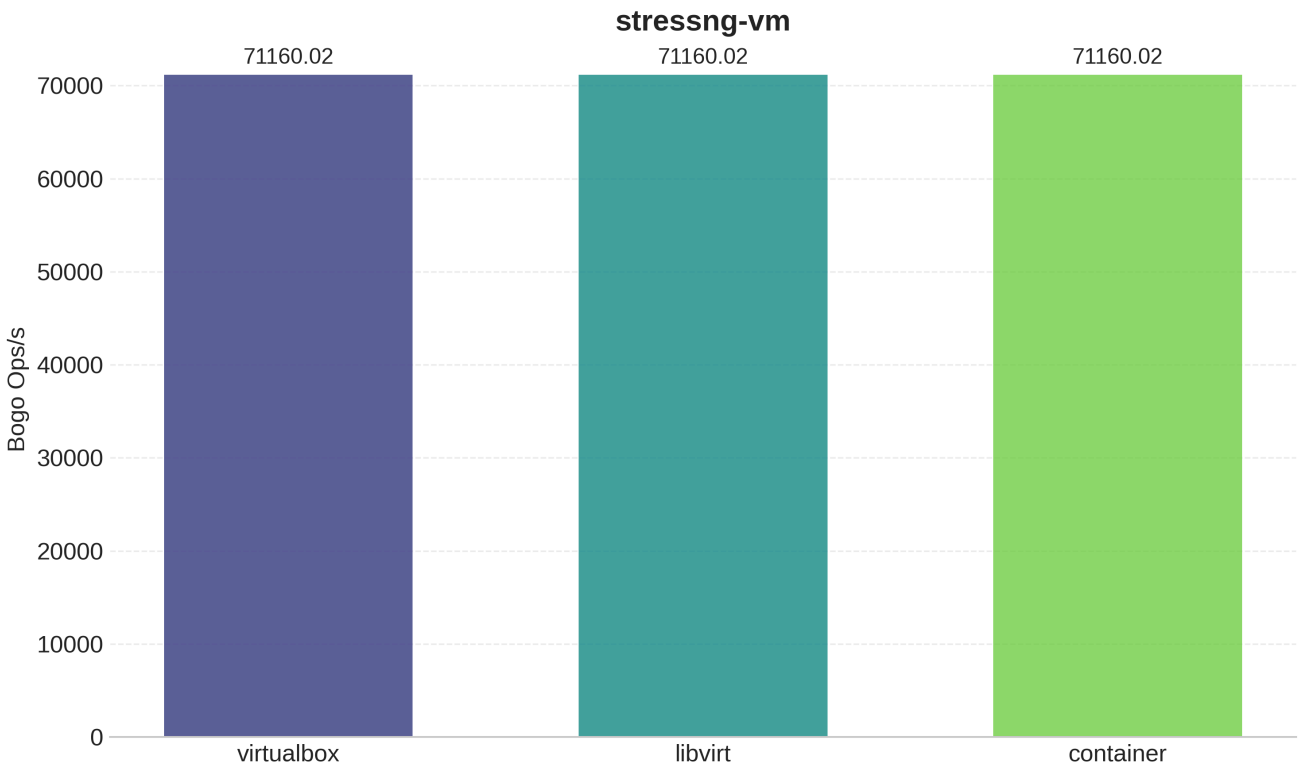
Based on the HPL benchmark performance graph comparing Docker and Libvirt, it is clear that Libvirt significantly outperforms Docker across all tested matrix sizes. Investigating the potential causes, system monitoring via `btop` showed a key difference in CPU usage: Docker containers distributed the HPL workload across all 12 available CPUs, while the Libvirt virtual machine achieved full (100%) utilization on a specific set of 4 cores. This suggests that the HPL benchmark on this system may benefit more from concentrating computational resources on fewer, tightly utilized cores rather than distributing the load widely, potentially due to factors like cache efficiency or inter-core communication overhead.



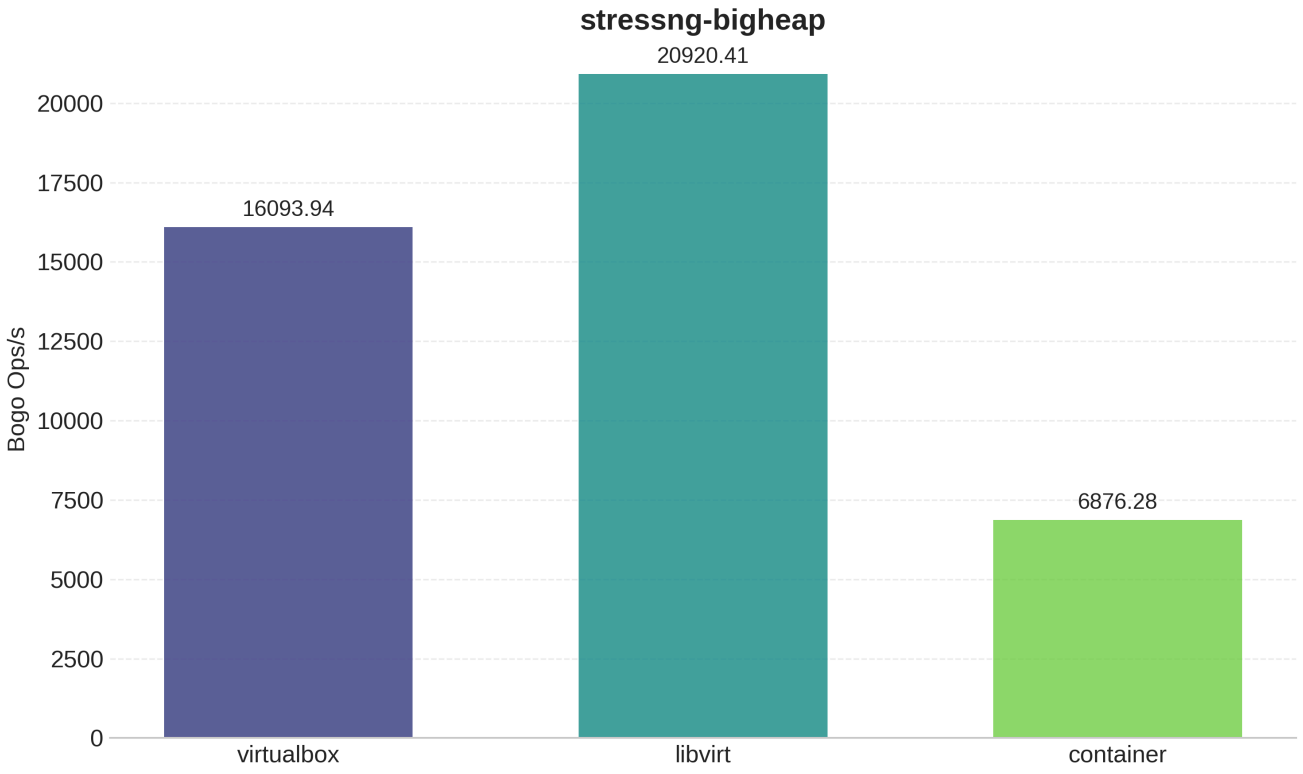


Memory

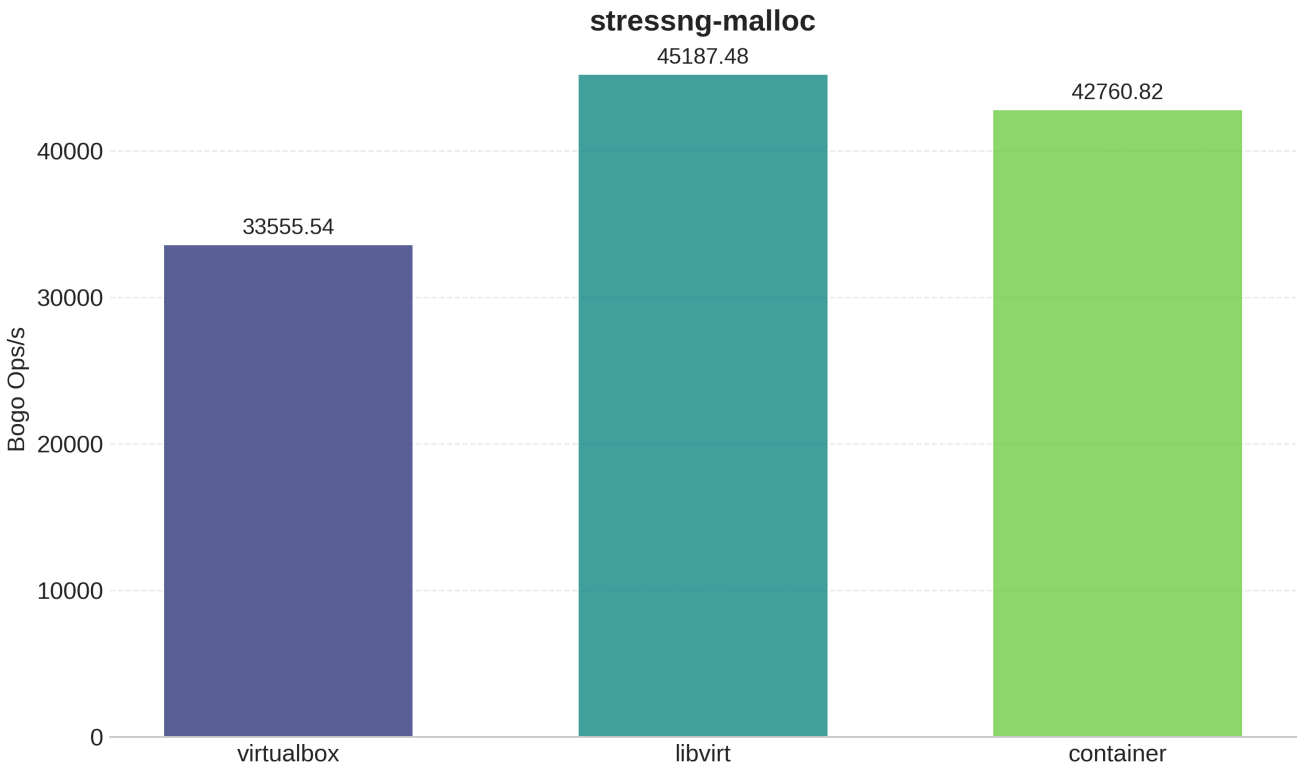
```
stress-ng --vm 4 --vm-bytes 1G --timeout 60s
```



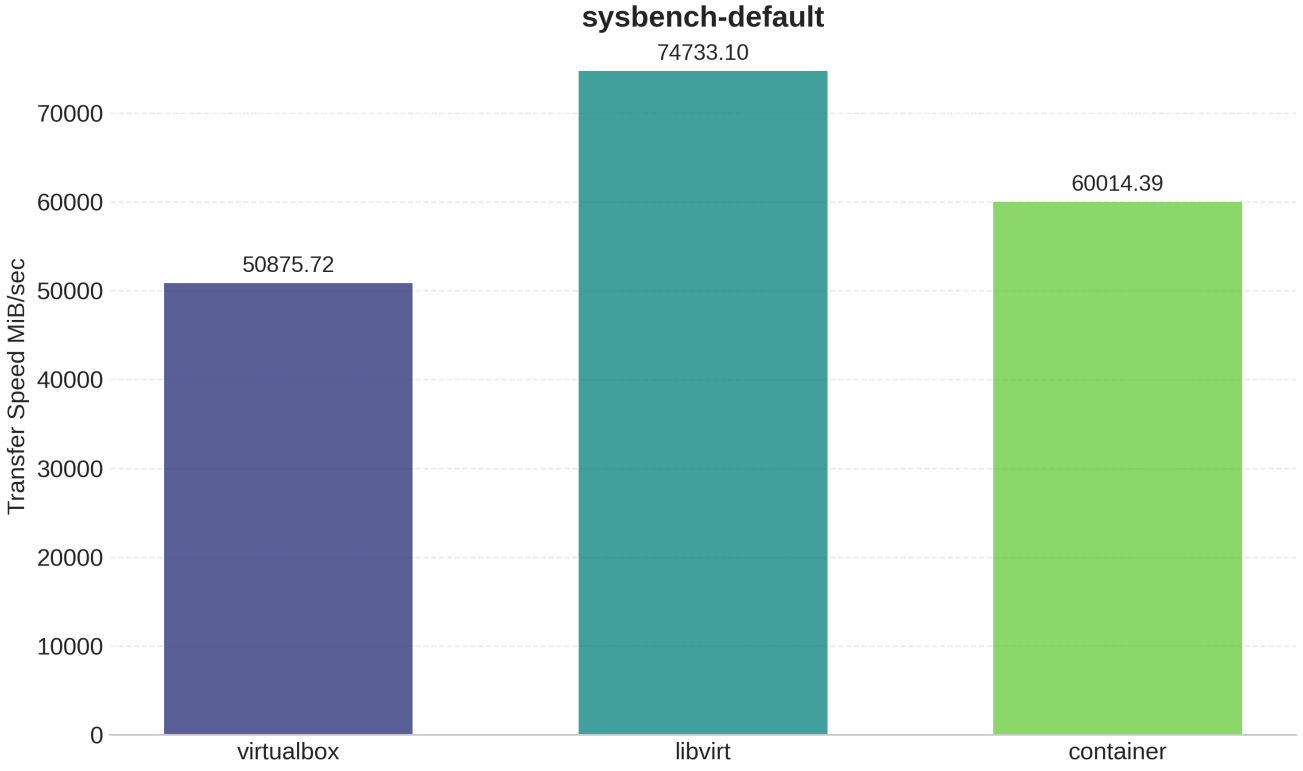

```
stress-ng --bigheap 2 --timeout 60s
```




```
stress-ng --malloc 4 --malloc-bytes 1G --timeout 60s
```




```
sysbench memory --memory-block-size=1M --memory-total-size=10G --threads=4
```

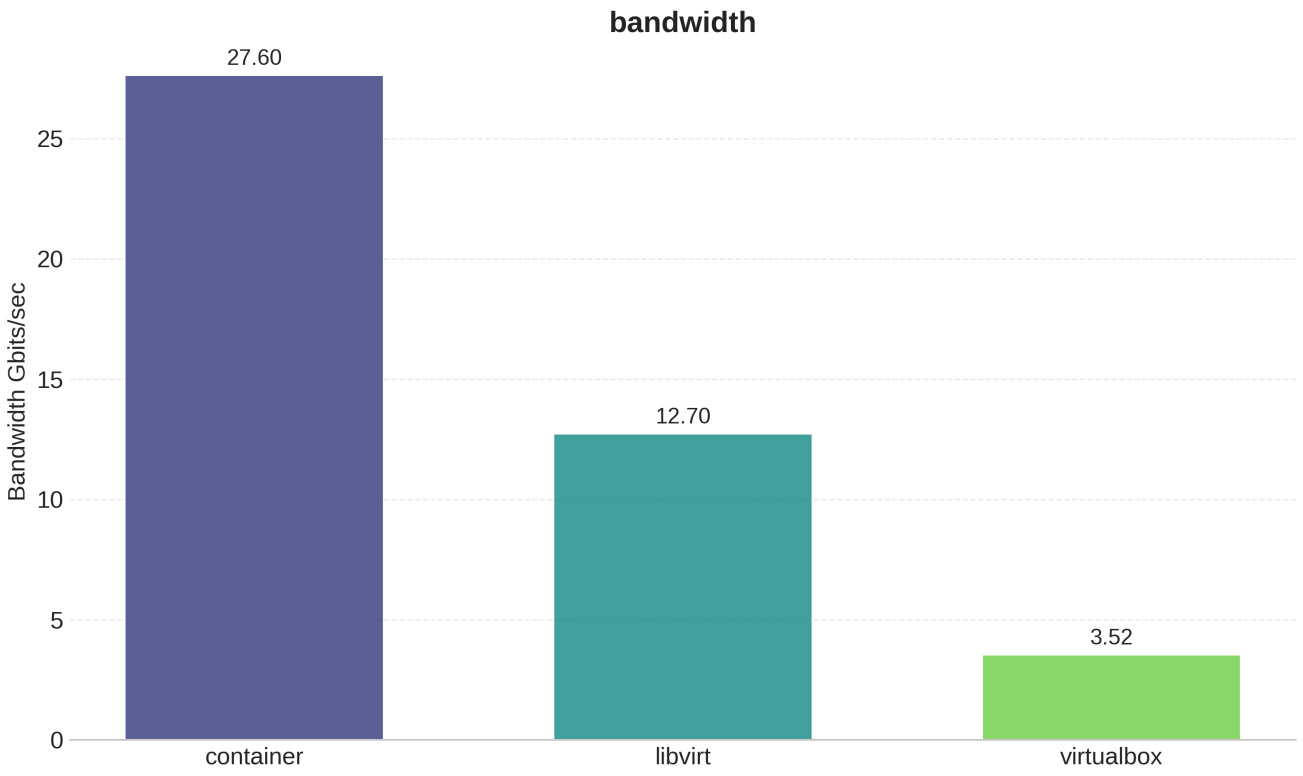


- In the `stressng-bigheap` test, which likely stresses large memory allocations, Libvirt demonstrated the highest performance, followed by VirtualBox, with the container showing significantly lower results.
- For `stressng-malloc`, a test focused on memory allocation and deallocation, Libvirt again performed the best, closely followed by the container, while VirtualBox had the lowest score.

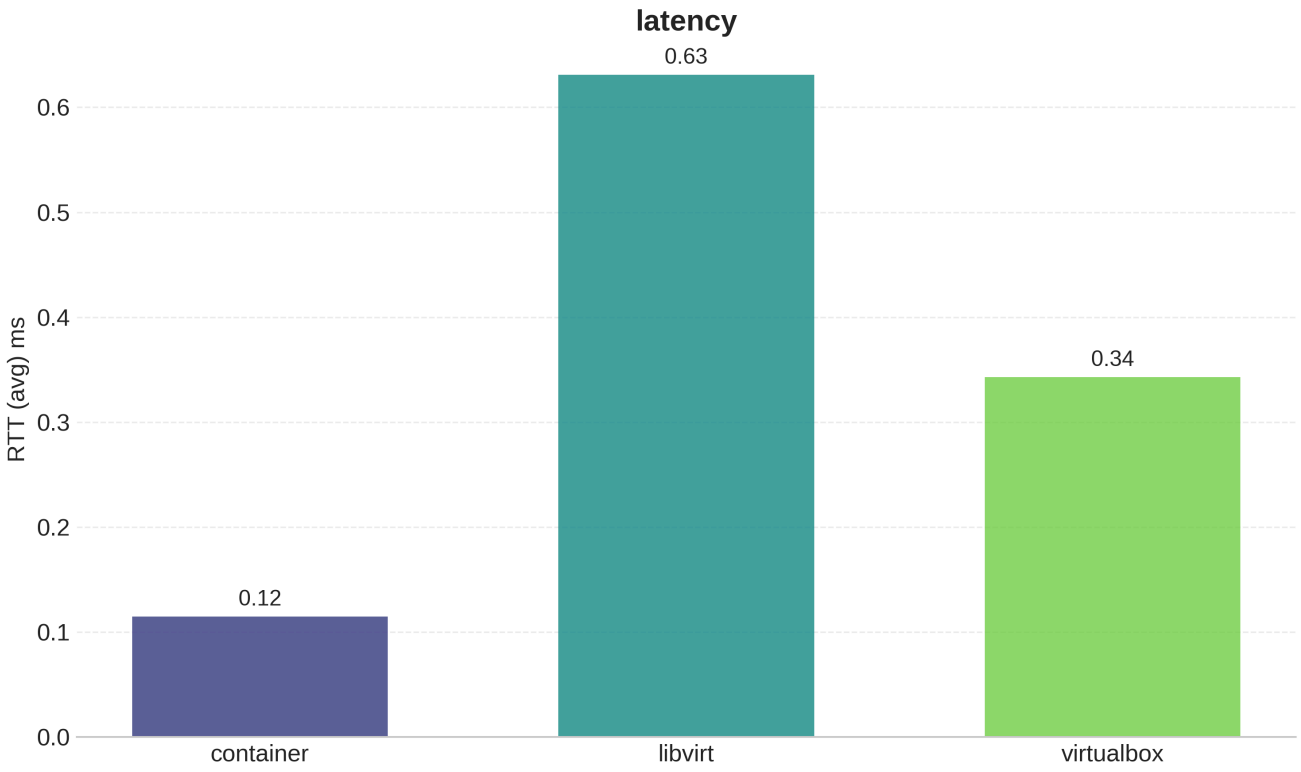
- The `stressng-vm` test, related to virtual memory operations, showed identical performance across all three environments: the container, Libvirt, and VirtualBox.
- For the `sysbench-default` test, which measures transfer speed, Libvirt again achieved the highest performance, followed by the container, and then VirtualBox.

Network

```
iperf --client iperf-server --time 30
```



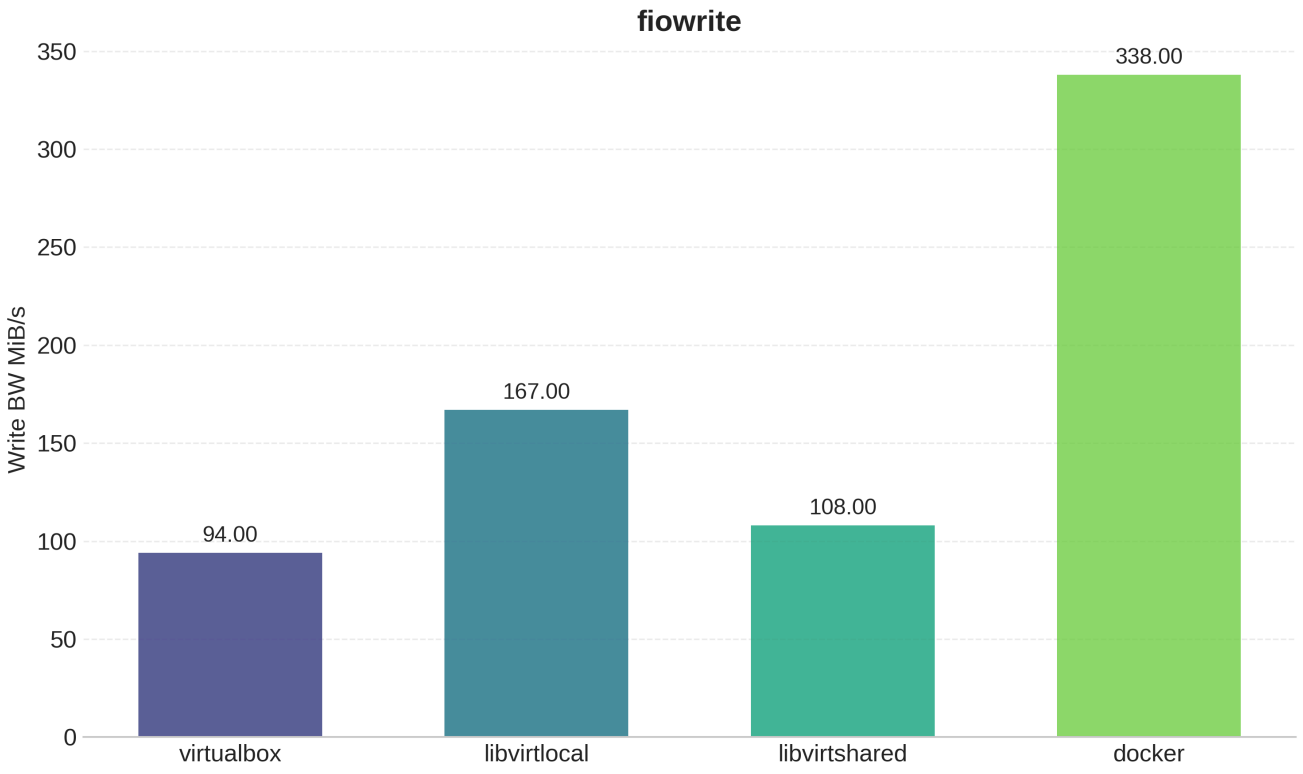

```
ping -c "50" -i 0.2 "TARGET_IP"
```



- **Container:** Shows significantly higher network bandwidth and lower latency.
- **Libvirt (KVM):** Exhibits good network bandwidth but surprisingly higher latency compared to both the container and VirtualBox in your test.
- **VirtualBox:** Has the lowest network bandwidth among the three but better latency than Libvirt (though not as good as the container).

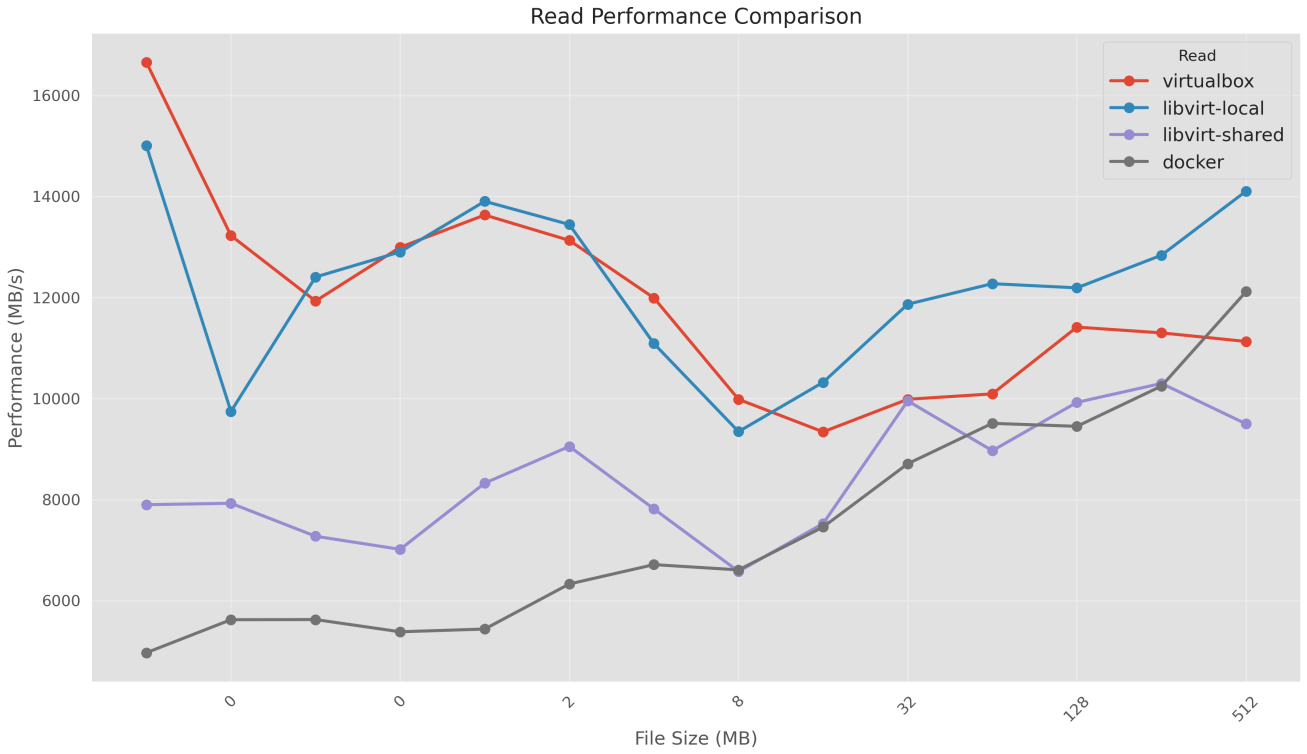
I/O

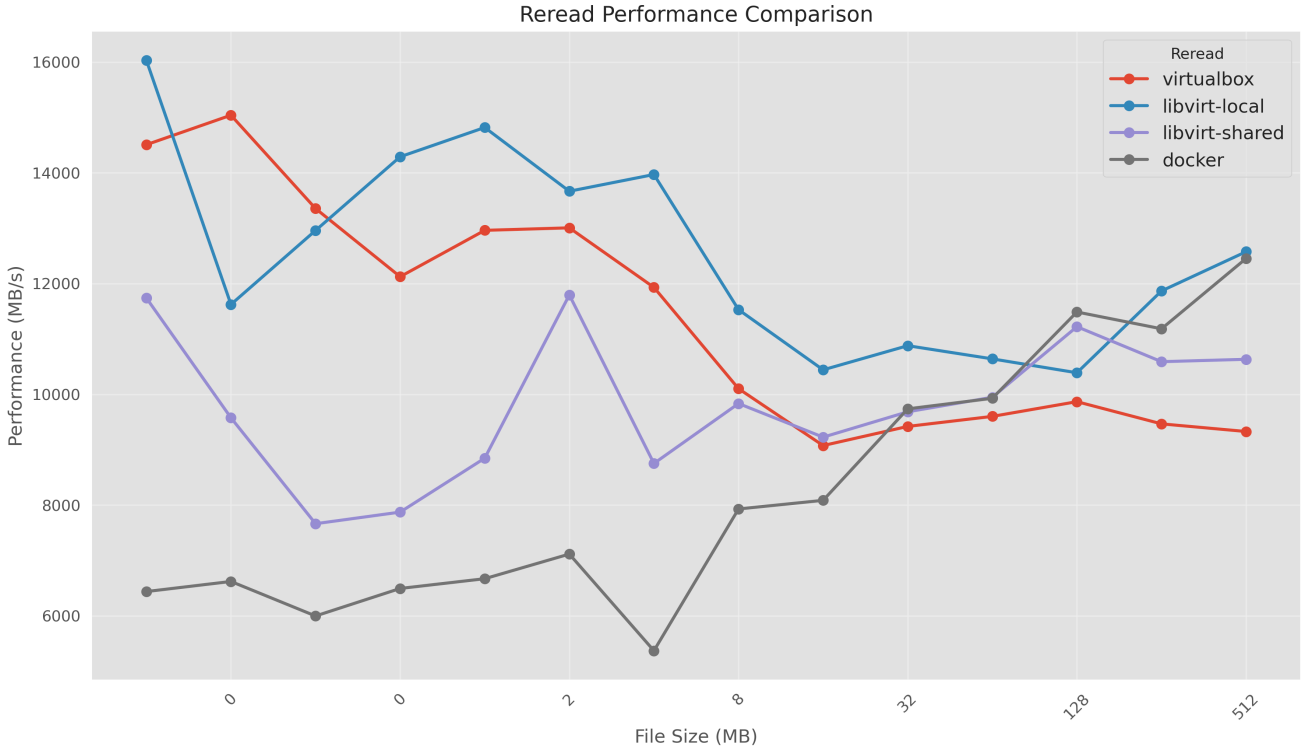
```
fio --name=mytest --ioengine=libaio --rw=randwrite --bs=4k --numjobs=16 \
    --size=1G --runtime=20s --time_based --unlink=1
```

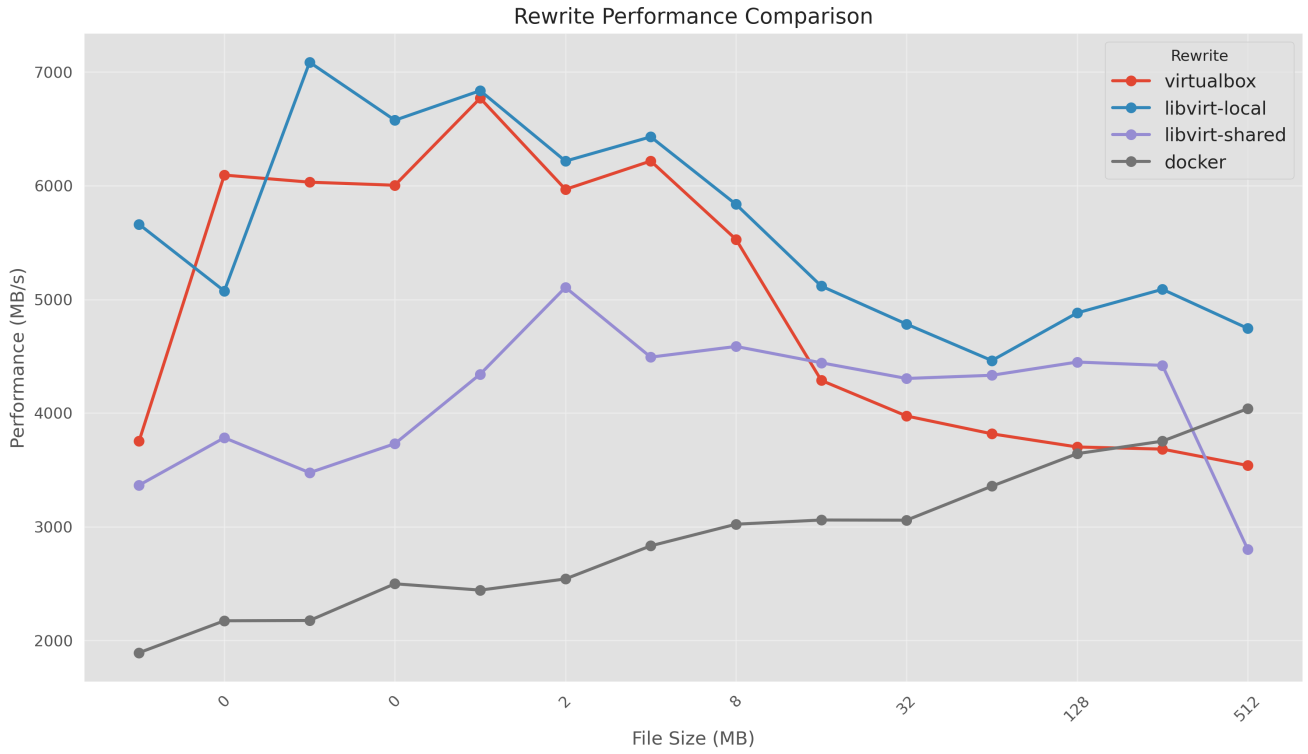


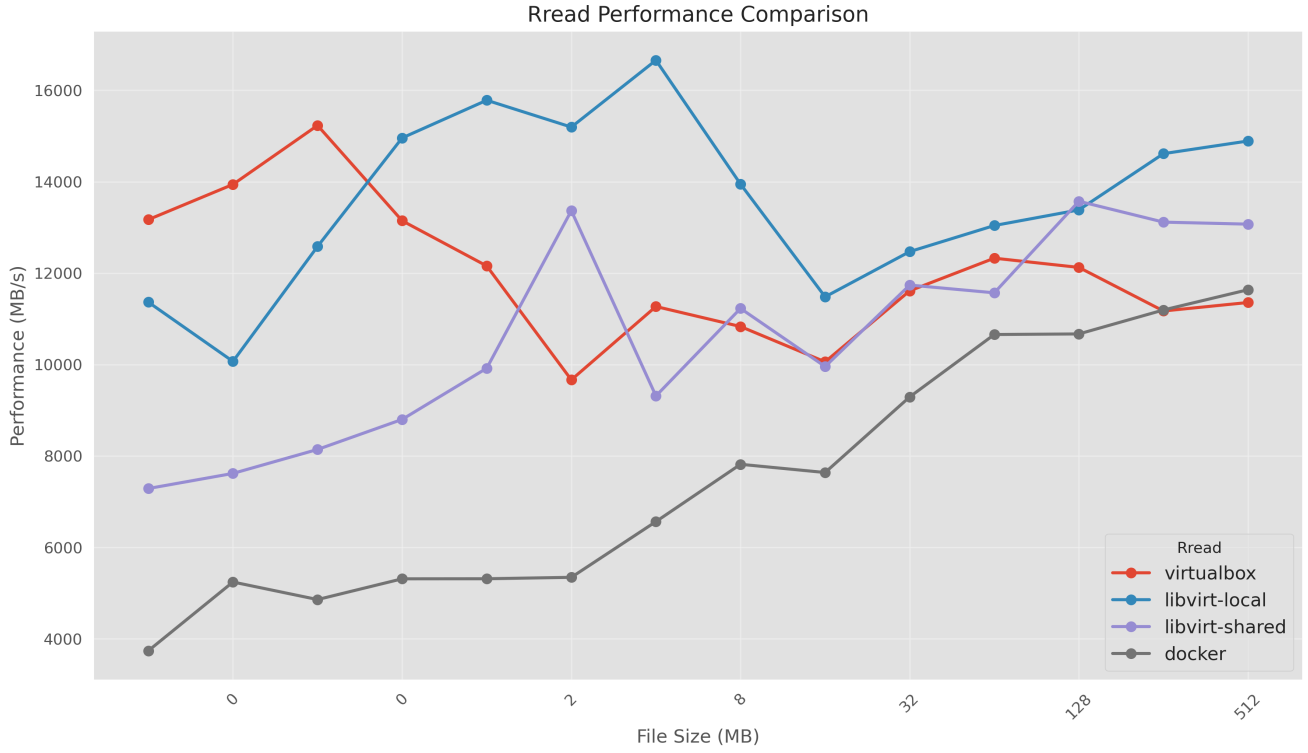
- **docker** demonstrates the highest write speed, reaching 338.00 MiB/s. This suggests superior I/O performance for write-intensive tasks within a docker container environment compared to the other options tested.
- **libvirtlocal** shows the second-best performance at 167.00 MiB/s, performing noticeably better than the shared storage configuration of libvirt.
- **libvirtshared** yields a write bandwidth of 108.00 MiB/s, indicating that using shared storage with libvirt in this scenario results in lower write performance compared to local storage or docker.
- **virtualbox** records the lowest write bandwidth among the four, at 94.00 MiB/s. This suggests it might be the least suitable option for applications requiring high write throughput based on this specific test.

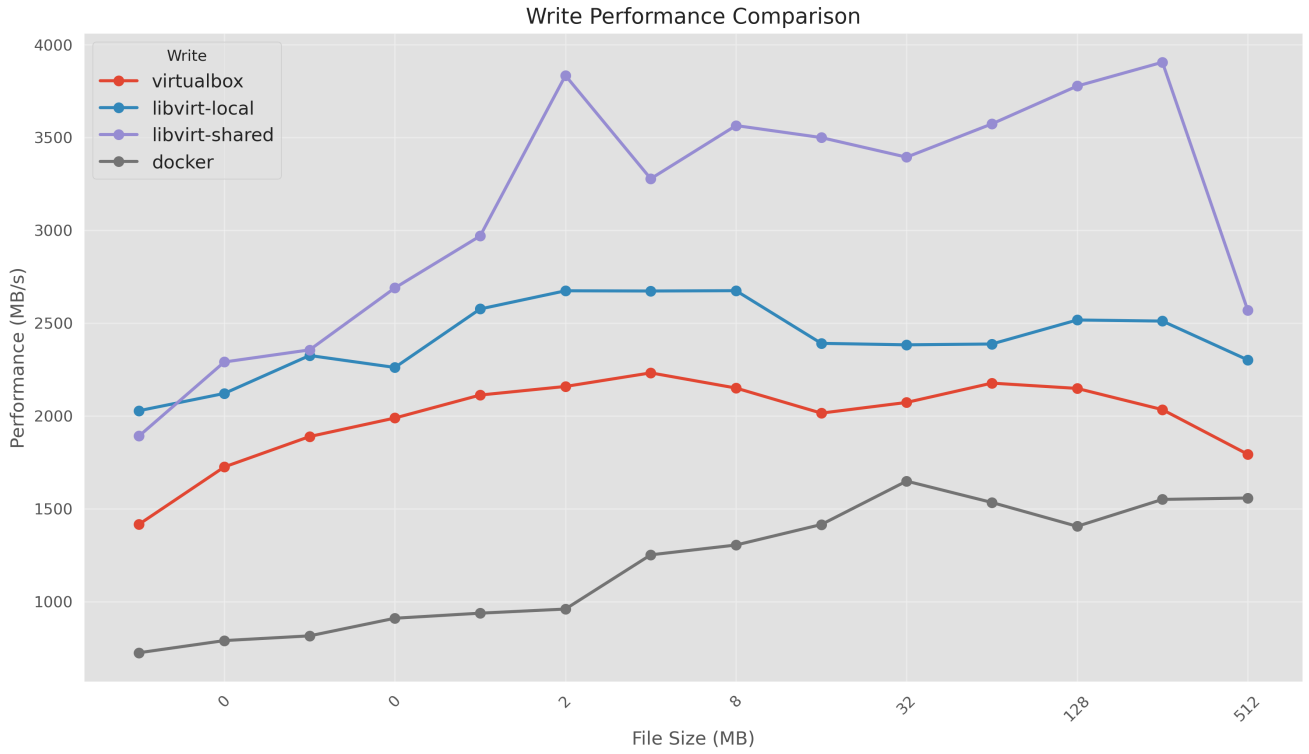
```
iozone -a
```


Read Performance Comparison
Performance (MB/s)
16000
14000
12000
10000
8000
6000
0
0
2
8
32
128
512
File Size (MB)
Read
virtualbox
libvirt-local
libvirt-shared
docker


Reread Performance Comparison
Performance (MB/s)
16000
14000
12000
10000
8000
6000
0
0
2
8
32
128
512
File Size (MB)
Reread
virtualbox
libvirt-local
libvirt-shared
docker


Rewrite Performance Comparison
Performance (MB/s)
7000
6000
5000
4000
3000
2000
0
0
2
8
32
128
512
File Size (MB)
Rewrite
virtualbox
libvirt-local
libvirt-shared
docker


Rread Performance Comparison
Performance (MB/s)
16000
14000
12000
10000
8000
6000
4000
0
0
2
8
32
128
512
File Size (MB)
Rread
virtualbox
libvirt-local
libvirt-shared
docker



- **libvirt-local** generally provides the best **read (sequential and random)** and **reread** performance across most file sizes compared to the other technologies. It also performs well for **rewrite** at smaller sizes.
- **libvirt-shared** stands out in **sequential write** performance, often surpassing `libvirt-local`, VirtualBox, and Docker. Its read performance is generally moderate.
- **virtualbox** shows inconsistent performance, sometimes performing well at very small file sizes, but often falling behind `libvirt-local` for read operations and `libvirt-shared` for write operations.
- **docker** consistently shows the lowest performance across all tested I/O operations and file sizes. This suggests that the overhead of Docker's storage handling in this setup might be less optimized for these specific block-level I/O benchmarks compared to the virtualization solutions.

Cloud Advanced

Network Interfaces Benchmark

Project Structure

```
.
├── assignment.md
├── README.md
└── results
    ├── bridge_bandwidth.txt
    ├── bridge_latency.txt
    ├── veth_bandwidth.txt
    └── veth_latency.txt
```

```
│   ├── vxlan_bandwidth.txt
│   └── vxlan_latency.txt
├── scripts
│   ├── run_test.sh
│   ├── setup
│   │   ├── create_namespaces.sh
│   │   ├── delete_bridge.sh
│   │   ├── delete_namespaces.sh
│   │   ├── delete_veth.sh
│   │   ├── delete_vxlan.sh
│   │   ├── setup_bridge.sh
│   │   ├── setup_veth.sh
│   │   └── setup_vxlan.sh
│   └── tests
│       ├── bandwidth_test.sh
│       └── latency_test.sh
└── Vagrantfile
```

Description

This project explores various methods for connecting two Linux network namespaces (`ns1` and `ns2`) and testing the network performance between them. The following connection methods are implemented and tested:

1. **Direct veth pair:** A single `veth` pair connects `ns1` directly to `ns2` .
2. **veth pairs connected to a single bridge:** Each namespace has a `veth` pair, and both pairs are connected to a single Linux bridge (`br0`).
3. **veth pairs on separate bridges connected via VXLAN:** Each namespace has a `veth` pair connected to its own bridge (`br0` for `ns1` , `br1` for `ns2`). These two bridges are then connected via a VXLAN tunnel.

The setup process involves creating the network namespaces, configuring the chosen connection method, assigning IP addresses, and ensuring network connectivity between `ns1` and `ns2` .

The scripts for setting up and tearing down these configurations are located in the `benchmark/setup` directory.

Network performance is measured between `ns1` and `ns2` for each connection method using `iperf` and `ping` . The key metrics collected are:

- **Bandwidth:** Measured using `iperf` .
- **Latency:** Measured using `ping` .
  The test scripts are located in the `benchmark/tests` directory. The main test execution script is `benchmark/run_test.sh` , which orchestrates the setup, testing, and deletion for each configuration.

Network Interfaces

Network Namespaces

A Linux **network namespace** provides an isolated copy of the network stack for a group of processes. This includes separate network interfaces, routing tables, IP addresses, firewall rules, and ports. Processes in one network namespace cannot see or interact with the network resources in another.

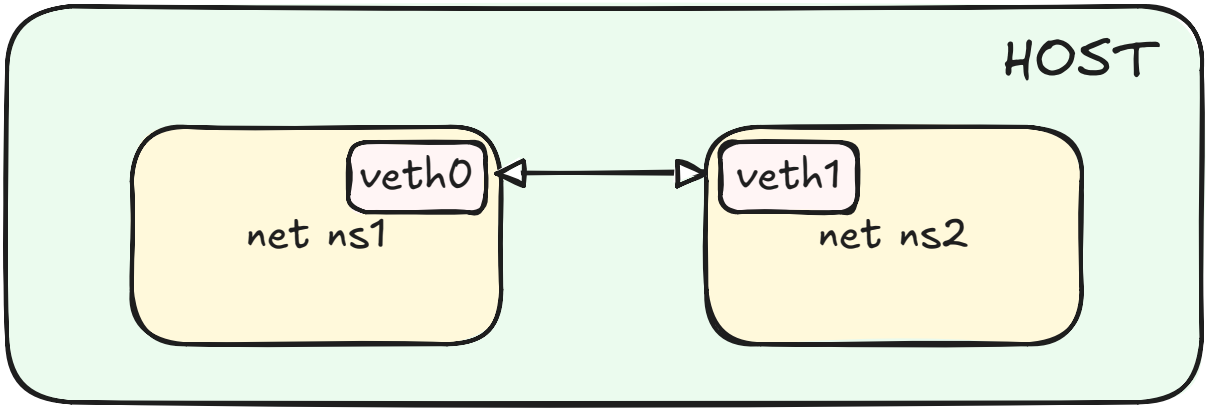
Each network namespace operates as if it were a completely separate machine from a networking perspective. For example, you can assign different IP addresses and routes in each namespace, making it useful for container networking, testing environments, or building custom virtual networks.

```
# Create network namespaces
echo "Creating network namespace ns1..."
sudo ip netns add ns1
echo "Network namespace ns1 created."

echo "Creating network namespace ns2..."
sudo ip netns add ns2
echo "Network namespace ns2 created."
```

Veth

A **veth** (virtual Ethernet) is a pair of connected virtual network interfaces used in Linux networking, particularly with **network namespaces** and **containers**.

Each **veth pair** acts like a virtual network cable: packets sent on one end are immediately received on the other. They are always created in pairs—for example, `veth0` and `veth1`. One end can be placed in one network namespace (such as a container), while the other remains in the host or another namespace. This forms a bridge between two network contexts.



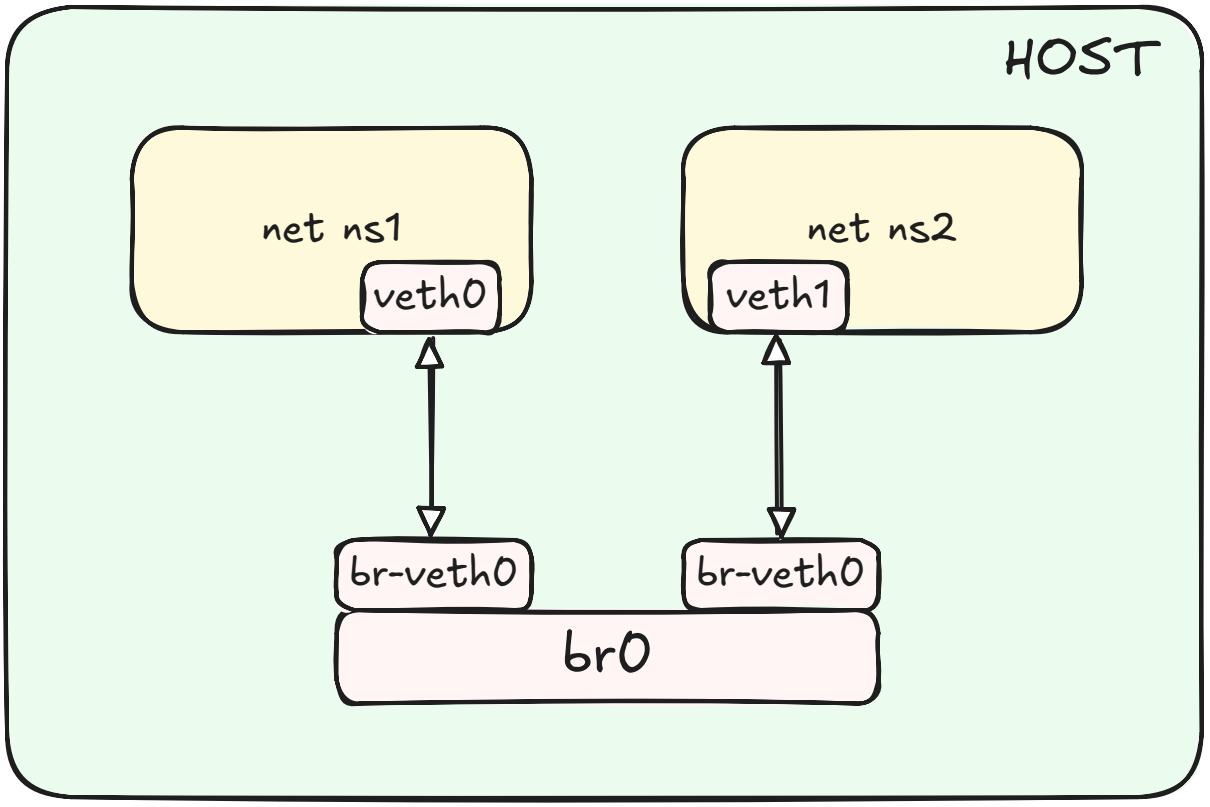
```
echo "Creating veth pair veth0/veth1..."
# Direct veth pair connection
ip link add veth0 type veth peer name veth1
echo "Veth pair created."

echo "Assigning veth interfaces to namespaces (veth0 to ns1, veth1 to
ns2)..."
ip link set veth0 netns ns1
ip link set veth1 netns ns2
echo "Veth interfaces assigned to namespaces."

echo "Configuring IP addresses for namespace interfaces..."
# Configure IP addresses
ip netns exec ns1 ip addr add 10.0.1.1/24 dev veth0
ip netns exec ns2 ip addr add 10.0.1.2/24 dev veth1
echo "IP addresses configured."

echo "Bringing up interfaces in namespaces (veth0 in ns1, veth1 and lo in
ns2)..."
# Bring up interfaces
ip netns exec ns1 ip link set veth0 up
ip netns exec ns1 ip link set lo up # Bringing up loopback in ns1
ip netns exec ns2 ip link set veth1 up
ip netns exec ns2 ip link set lo up # Bringing up loopback in ns2
echo "Interfaces are up in namespaces."
```

Bridge

A **bridge** in Linux networking is a virtual device that connects multiple network interfaces at the **data link layer (Layer 2)**, similar to a physical Ethernet switch. It forwards packets between interfaces based on MAC addresses, allowing devices on those interfaces to communicate as if they were on the same physical network.

Bridges are commonly used in conjunction with **veth pairs** and **network namespaces** to connect isolated network environments (like containers) to each other or to the host network.

For example, a common setup involves creating a bridge (e.g., `br0`), connecting the host's physical network interface and the host end of a `veth` pair to the bridge, and moving the other end of the `veth` pair into a network namespace. This enables the namespace to communicate with the outside network via the bridge.

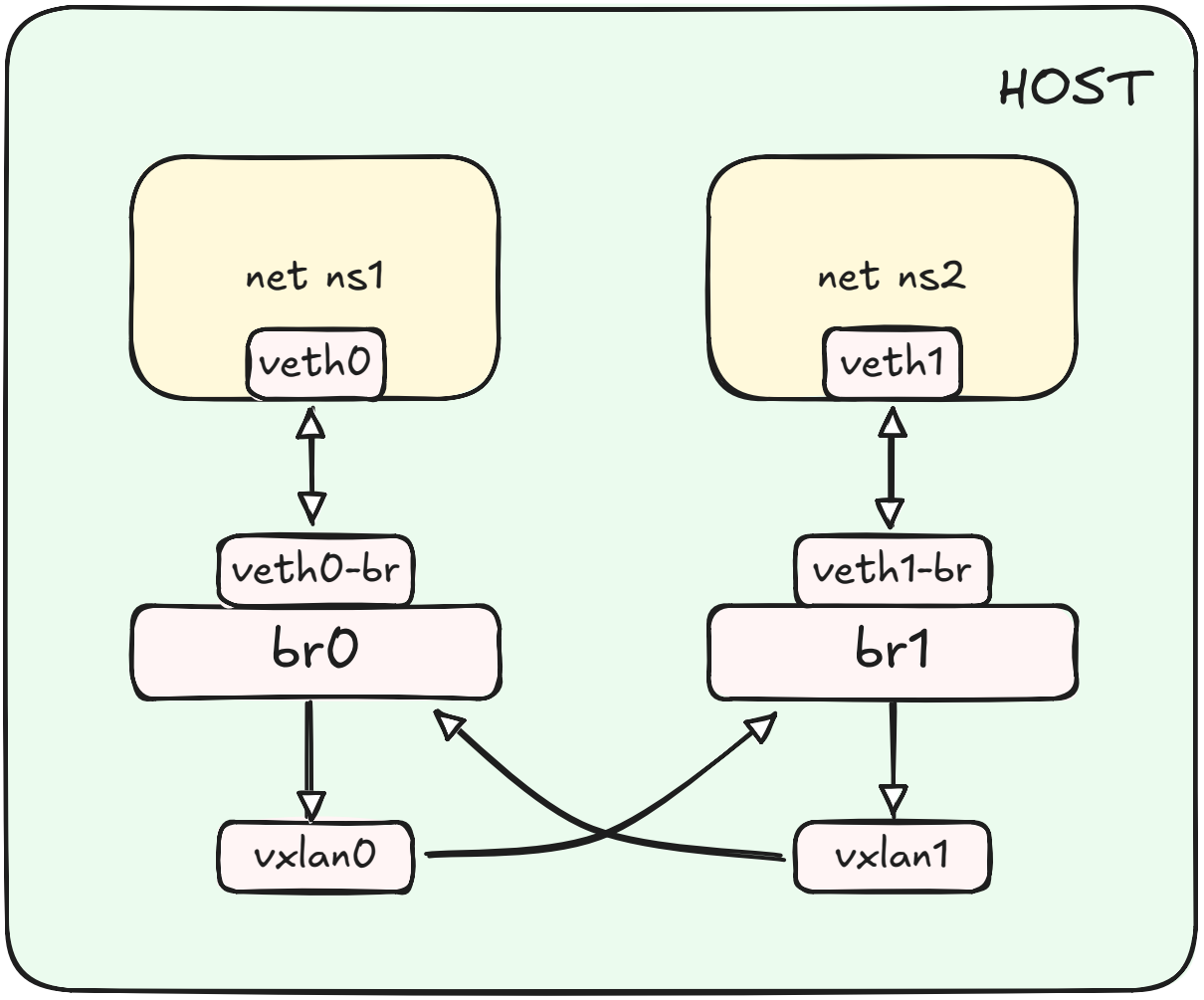



```
echo "Creating bridge br0..."
# Create bridge
ip link add br0 type bridge
echo "Bridge br0 created."

echo "Bringing up bridge br0..."
ip link set br0 up
echo "Bridge br0 is up."

echo "Creating veth pairs veth0/br-veth0 and veth1/br-veth1..."
# Create veth pairs
ip link add veth0 type veth peer name br-veth0
ip link add veth1 type veth peer name br-veth1
echo "Veth pairs created."

echo "Connecting veth pairs to namespaces (ns1, ns2) and bridge (br0)..."
# Connect veth pairs to namespaces and bridge
ip link set veth0 netns ns1
ip link set veth1 netns ns2
ip link set br-veth0 master br0
ip link set br-veth1 master br0
echo "Veth pairs connected."

echo "Bringing up bridge-side veth interfaces (br-veth0, br-veth1)..."
ip link set br-veth0 up
ip link set br-veth1 up
echo "Bridge-side veth interfaces are up."
```

```
echo "Configuring IP addresses for namespace-side veth interfaces..."
# Configure IP addresses
ip netns exec ns1 ip addr add 10.0.2.1/24 dev veth0
ip netns exec ns2 ip addr add 10.0.2.2/24 dev veth1
echo "IP addresses configured."

echo "Bringing up namespace-side veth interfaces (veth0 in ns1, veth1 in
ns2)..."
# Bring up interfaces
ip netns exec ns1 ip link set veth0 up
ip netns exec ns2 ip link set veth1 up
echo "Namespace-side veth interfaces are up."

echo "Network setup script completed successfully!"
```

VXLAN

VXLAN (Virtual Extensible LAN) is a network virtualization technology designed to address the limitations of traditional VLANs in large-scale cloud and data center environments. It allows you to create **Layer 2 overlay networks** on top of **Layer 3 infrastructure**, enabling communication between isolated networks across different physical locations.



```
echo "Creating bridges..."
ip link add br0 type bridge
ip link add br1 type bridge
ip link set br0 up
ip link set br1 up
echo "Bridges created and activated."

echo "Creating and configuring veth pairs for ns1..."
ip link add veth0 type veth peer name veth0-br
ip link set veth0-br up
ip link set veth0-br master br0
ip link set veth0 netns ns1
ip netns exec ns1 ip link set veth0 up
ip netns exec ns1 ip addr add 10.1.1.2/24 dev veth0
echo "ns1 veth pair configured."

echo "Creating and configuring veth pairs for ns2..."
ip link add veth1 type veth peer name veth1-br
ip link set veth1-br up
ip link set veth1-br master br1
ip link set veth1 netns ns2
ip netns exec ns2 ip link set veth1 up
ip netns exec ns2 ip addr add 10.1.2.2/24 dev veth1
echo "ns2 veth pair configured."

echo "Setting up VXLAN tunnel between bridges..."
ip link add vxlan0 type vxlan id 100 local 127.0.0.1 remote 127.0.0.1
dstport 4789 dev lo
ip link add vxlan1 type vxlan id 101 local 127.0.0.1 remote 127.0.0.1
dstport 4790 dev lo
ip link set vxlan0 master br0
ip link set vxlan1 master br1
ip link set vxlan0 up
ip link set vxlan1 up
echo "VXLAN tunnel established."

echo "Configuring IP addresses and routing..."
ip addr add 10.1.1.1/24 dev br0
ip addr add 10.1.2.1/24 dev br1
ip netns exec ns1 ip route add 10.1.2.0/24 via 10.1.1.1
ip netns exec ns2 ip route add 10.1.1.0/24 via 10.1.2.1
echo "IP addressing and routing configured."

echo "Enabling IP forwarding..."
sysctl -w net.ipv4.ip_forward=1
echo "IP forwarding enabled."
```

```
echo "Network setup with VXLAN completed successfully!"
```

Managing the vm

In order to ensure reproducibility the experiment is done in a virtualized environment.

Spin up the VM

```
> vagrant up
```

Halt the VM

```
> vagrant halt
```

Destroy the VM

```
> vagrant destroy
```

Benchmarking

Access the VM and execute the tests:

```
> vagrant ssh vm
vagrant@vm:~$ cd scripts
vagrant@vm:~/scripts$ ./run_test.sh
```

The `run_test.sh` script will execute the setup, testing, and deletion for each connection method. This process will take some time to complete.

Exit the VM:

```
vagrant@vm:~$ logout
```

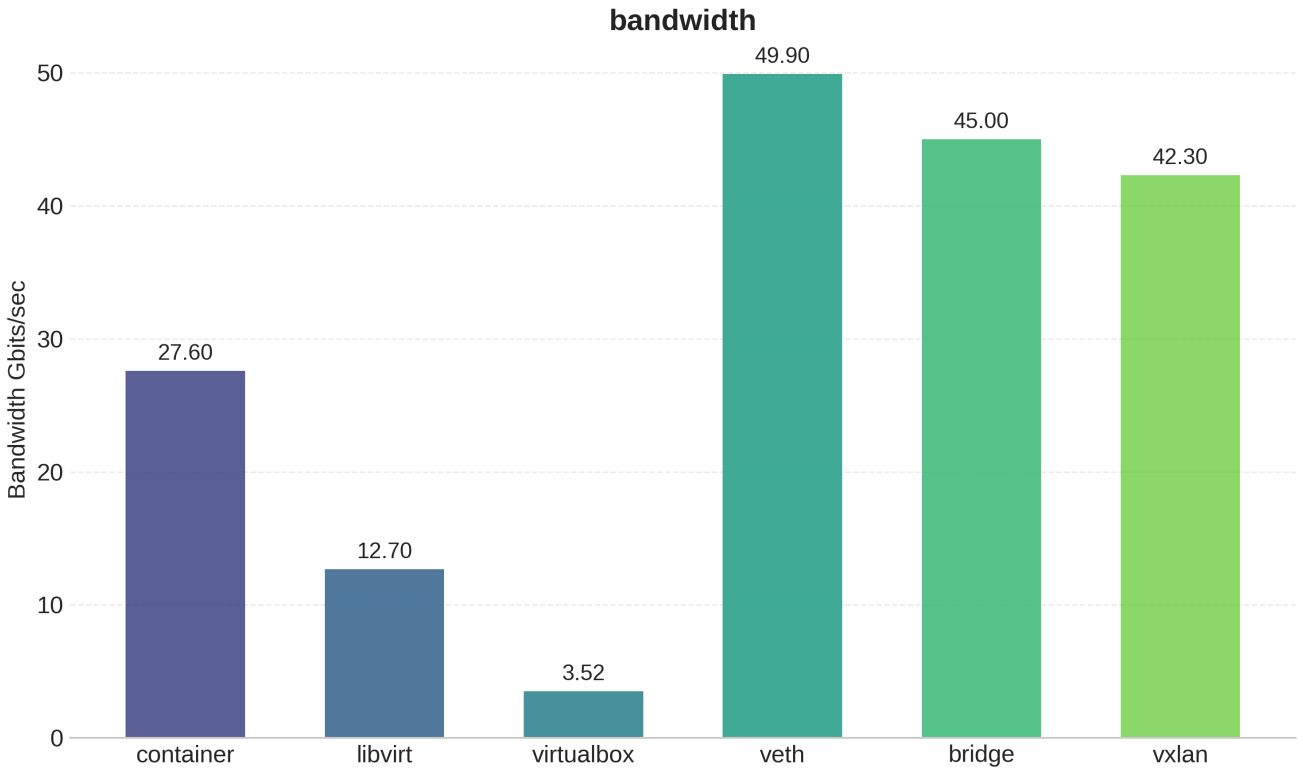
Retrieve the results:

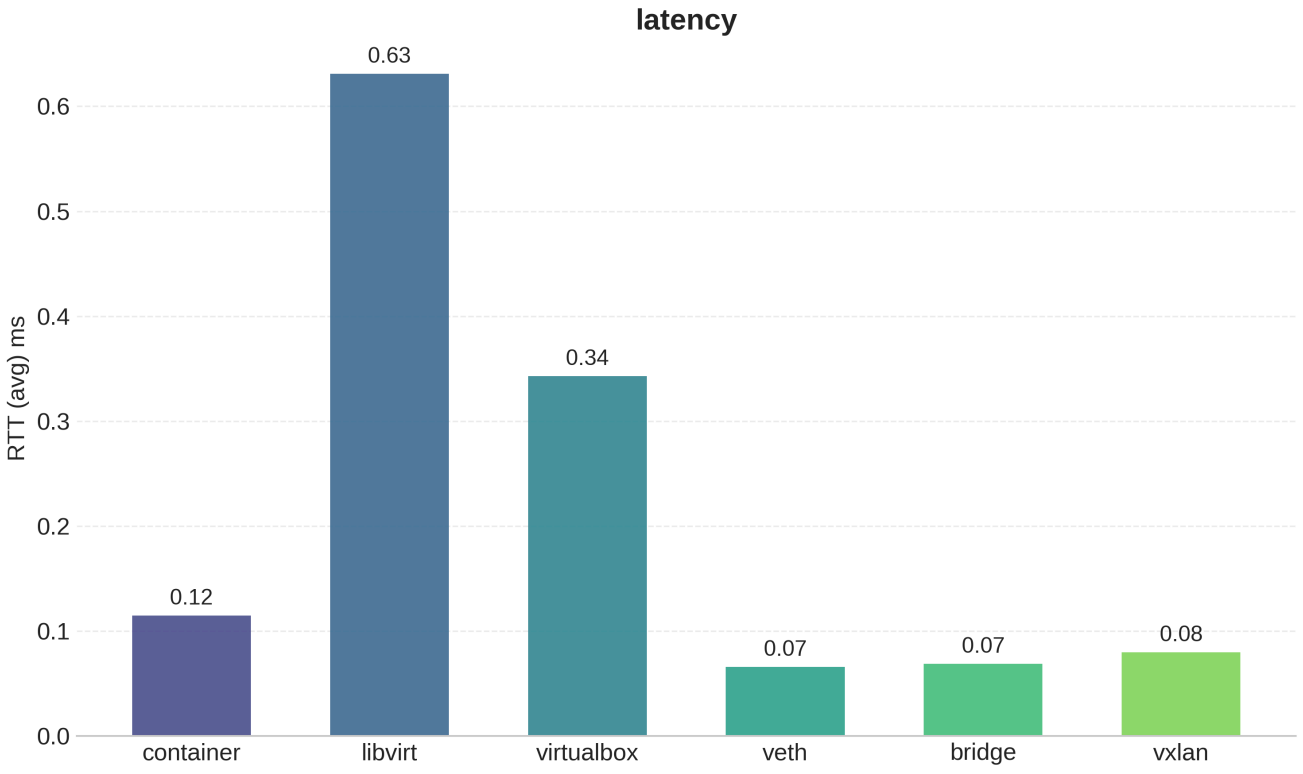
```
> vagrant scp vm:/home/vagrant/results .
```

This command will copy the `results` directory from the VM to your local machine.

Results

Test Type	Bandwidth	Latency (avg rtt)
VETH	49.9 Gbits/sec	0.066 ms
Bridge	45.0 Gbits/sec	0.069 ms
VXLAN	42.3 Gbits/sec	0.080 ms





- **Highest Bandwidth / Lowest Latency:** `veth` and `bridge` configurations generally offer the best network performance in terms of both high bandwidth and low latency.

`vxlan` is also very performant, only slightly behind.

- **Container Networking:** The 'container' configuration shows good bandwidth but slightly higher latency compared to the raw `veth` / `bridge` / `vxlan` links.
- **Virtualization Networking Overhead:** Both `libvirt` and `virtualbox` introduce significant overhead compared to container networking and the underlying network constructs.
- **VirtualBox Performance:** VirtualBox shows particularly poor network performance in this benchmark, with the lowest bandwidth and the second-highest latency.
- **Libvirt Performance:** Libvirt's networking performance is better than VirtualBox's in terms of bandwidth but exhibits the highest latency among all tested categories.
-

Kubernetes

Project Structure

```
├── assignment.md
├── hpl
│   ├── Dockerfile
│   ├── hpl-benchmark.tar
│   └── HPL.dat
├── kubernetes-setup
│   ├── join-command
│   ├── k8s-playbook.yaml
│   └── playground_kubeconfig.yaml
├── README.md
└── Vagrantfile
```

Description

Vagrantfile

This Vagrantfile is designed to create a virtual environment for setting up a Kubernetes cluster.

1. **Defines Base Configuration:** It sets the base box to `generic/ubuntu2004` and allocates 4096MB of memory and 4 CPUs to each virtual machine.
2. **Lists Servers:** It defines an array `servers` containing details for three virtual machines: `kube-master` (192.168.50.10), `kube-01` (192.168.50.11), and `kube-02` (192.168.50.12).
3. **Configures Provider:** It specifies the `libvirt` provider for virtualization and sets some provider-specific options.

4. **Sets up Ansible Provisioning:** It configures Vagrant to run an Ansible playbook (`kubernetes-setup/k8s-playbook.yaml`) after the VMs are created. It also defines Ansible groups `control` (containing `kube-master`) and `workers` (containing `kube-01` and `kube-02`) for the playbook to target.
5. **Defines Individual VMs:** It iterates through the `servers` array, defining each virtual machine with its specified hostname and configuring a private network interface with its assigned IP address.

```
VAGRANT_BOX = "generic/ubuntu2004"
VM_MEMORY = 4096
VM_CPUS = 4

servers = [
  { :hostname => "kube-master", :ip => "192.168.50.10" },
  { :hostname => "kube-01", :ip => "192.168.50.11" },
  { :hostname => "kube-02", :ip => "192.168.50.12" },
]

Vagrant.configure("2") do |config|
    config.vm.box = VAGRANT_BOX

    config.vm.provider "libvirt" do |lv|
        lv.memory = VM_MEMORY
        lv.cpus = VM_CPUS
        lv.qemu_use_session = false
    end

    # Define the main Ansible provisioner block
    config.vm.provision "ansible" do |ansible|
        ansible.playbook = "kubernetes-setup/k8s-playbook.yaml"

        # Define Ansible groups
        ansible.groups = {
            "control" => ["kube-master"],
            "workers" => servers.select { |s| s[:hostname] != "kube-
master" }.map { |s| s[:hostname] },
        }
    end

    # Define individual machines
    servers.each do |conf|
        config.vm.define conf[:hostname] do |node|
            node.vm.hostname = conf[:hostname]
            node.vm.network "private_network", ip:conf[:ip]
        end
    end
```

```
end
```

Ansible playbook

Common provisioning

1. **Play Definition:**

```
- hosts: all
  become: true
  vars:
    # Kubernetes repository settings
    k8s_version: "v1.32"  # Update this when upgrading Kubernetes
    k8s_url_apt_key: "https://pkgs.k8s.io/core:/stable:/{{ k8s_version
}}/deb/Release.key"
    k8s_gpgpath: "/usr/share/keyrings/kubernetes-apt-keyring.gpg"
    k8s_repository: "https://pkgs.k8s.io/core:/stable:/{{ k8s_version
}}/deb/"

    # Network configuration
    pod_network_cidr: "10.244.0.0/16"  # CIDR for Flannel
```

   - `- hosts: all` : This line specifies that this play should run on all hosts defined in the Ansible inventory.
   - `become: true` : This indicates that Ansible should execute all tasks within this play with elevated privileges (equivalent to using `sudo`).
   - `vars:` : This section defines variables that can be used throughout this play.
2. **Variable Definitions:**

```
  vars:
    # Kubernetes repository settings
    k8s_version: "v1.32"  # Update this when upgrading Kubernetes
    k8s_url_apt_key: "https://pkgs.k8s.io/core:/stable:/{{ k8s_version
}}/deb/Release.key"
    k8s_gpgpath: "/usr/share/keyrings/kubernetes-apt-keyring.gpg"
    k8s_repository: "https://pkgs.k8s.io/core:/stable:/{{ k8s_version
}}/deb/"

    # Network configuration
    pod_network_cidr: "10.244.0.0/16"  # CIDR for Flannel
```

   - `k8s_version: "v1.32"` : Defines a variable `k8s_version` to hold the desired Kubernetes version string. This makes it easy to update the version later.

- `k8s_url_apt_key: "https://pkgs.k8s.io/core:/stable:/{{ k8s_version }}/deb/Release.key"` : Defines a variable `k8s_url_apt_key` constructed using the `k8s_version` variable, pointing to the URL for the Kubernetes APT repository signing key.
- `k8s_gpgpath: "/usr/share/keyrings/kubernetes-apt-keyring.gpg"` : Defines the target path on the remote hosts where the downloaded GPG key will be stored.
- `k8s_repository: "https://pkgs.k8s.io/core:/stable:/{{ k8s_version }}/deb/"` : Defines the base URL for the Kubernetes APT repository, also using the `k8s_version` variable.
- `pod_network_cidr: "10.244.0.0/16"` : Defines a variable `pod_network_cidr` to specify the network range that will be used for Kubernetes pods, commonly used by network plugins like Flannel.

3. **Tasks:**

```
- name: Install necessary system packages
  apt:
    name: "{{ packages }}"
    state: present
    update_cache: yes
  vars:
    packages:
      - apt-transport-https
      - ca-certificates
      - curl
      - software-properties-common
      - gnupg2
      - net-tools
      - jq
```

Install necessary system packages: Uses the `apt` module to install a list of essential system packages required for the setup. The list of packages is defined in a task-specific `vars` section. `update_cache: yes` ensures the package list is current.

```
- name: Install containerd
  apt:
    name: containerd
    state: present
    update_cache: yes
```

Install containerd_: Uses the `apt` module to install the `containerd` package, which is a standard container runtime used by Kubernetes.

```
- name: Restart containerd service
  systemd:
    name: containerd
```

```
    state: restarted
    enabled: true
```

Restart containerd service: Uses the `systemd` module to ensure the `containerd` service is restarted and enabled to start automatically on boot.

```
- name: Configure required kernel modules
  blockinfile:
    create: true
    path: /etc/modules-load.d/containerd.conf
    block: |
      overlay
      br_netfilter
```

Configure required kernel modules: Uses the `blockinfile` module to add the `overlay` and `br_netfilter` kernel modules to `/etc/modules-load.d/containerd.conf`. These modules are necessary for Kubernetes networking and storage. `create: true` ensures the file is created if it doesn't exist.

```
- name: Load kernel modules immediately
  shell: |
    modprobe overlay
    modprobe br_netfilter
```

Load kernel modules immediately: Uses the `shell` module to execute `modprobe` commands, loading the `overlay` and `br_netfilter` kernel modules into the running kernel immediately.

```
- name: Remove swap entries from /etc/fstab
  mount:
    name: "{{ item }}"
    fstype: swap
    state: absent
  with_items:
    - swap
    - none
```

Remove swap entries from /etc/fstab: Uses the `mount` module with `state: absent` to remove any lines related to swap partitions in `/etc/fstab`. Kubernetes requires swap to be disabled. The `with_items` loop iterates over common swap entry names.

```
- name: Disable swap immediately
  command: swapoff -a
  when: ansible_swaptotal_mb > 0
```

Disable swap immediately: Uses the `command` module to execute `swapoff -a`, disabling swap on the system immediately. The `when: ansible_swaptotal_mb > 0` condition ensures this command only runs if swap is currently active.

```
  - name: Configure kernel parameters for Kubernetes networking
    blockinfile:
      create: true
      path: /etc/sysctl.d/99-kubernetes-cri.conf
      block: |
        net.bridge.bridge-nf-call-iptables = 1
        net.bridge.bridge-nf-call-ip6tables = 1
        net.ipv4.ip_forward = 1
```

Configure kernel parameters for Kubernetes networking: Uses the `blockinfile` module to add specific network-related kernel parameters to `/etc/sysctl.d/99-kubernetes-cri.conf`. These settings are crucial for Kubernetes networking, particularly for how the kernel handles bridged network traffic and IP forwarding.

```
- name: Apply sysctl parameters without reboot
  command: sysctl --system
```

Apply sysctl parameters without reboot: Uses the `command` module to execute `sysctl --system`, which applies the kernel parameters defined in `/etc/sysctl.d/` without requiring a system reboot.

```
  - name: Add Kubernetes GPG key
    shell: curl -fsSL {{ k8s_url_apt_key }} | gpg --dearmor -o {{
k8s_gpgpath }}
    args:
      creates: "{{ k8s_gpgpath }}"
```

Add Kubernetes GPG key: Uses the `shell` module to download the Kubernetes APT repository GPG key using `curl`, dearmor it using `gpg`, and save it to the path specified by the `k8s_gpgpath` variable. The `args: creates: "{{ k8s_gpgpath }}"` ensures this task is skipped if the key file already exists.

```
  - name: Add Kubernetes repository
    apt_repository:
      filename: kubernetes
      repo: "deb [signed-by={{ k8s_gpgpath }}] {{ k8s_repository }} /"
```

Add Kubernetes repository: Uses the `apt_repository` module to add the Kubernetes APT repository configuration to the system's sources list. It uses the `k8s_gpgpath`

variable to specify the signing key and the `k8s_repository` variable for the repository URL.

```
  - name: Install Kubernetes components
    apt:
      name: "{{ packages }}"
      state: present
      update_cache: yes
    vars:
      packages:
        - kubelet
        - kubeadm
        - kubectl
```

Install Kubernetes components: Uses the `apt` module to install the core Kubernetes components: `kubelet` (the node agent), `kubeadm` (the cluster bootstrapping tool), and `kubectl` (the command-line tool for interacting with clusters). The package names are defined in a task-specific `vars` section.

```
  - name: Hold Kubernetes packages at installed version
    dpkg_selections:
      name: "{{ item }}"
      selection: hold
    loop:
      - kubelet
      - kubeadm
      - kubectl
```

Hold Kubernetes packages at installed version: Uses the `dpkg_selections` module to set the package selection for `kubelet`, `kubeadm`, and `kubectl` to `hold`. This prevents these packages from being automatically updated by `apt` commands, ensuring version consistency. The `loop` iterates over the list of package names.

```
  - name: Enable and start kubelet service
    systemd:
      name: kubelet
      daemon_reload: true
      state: started
      enabled: true
```

Enable and start kubelet service: Uses the `systemd` module to manage the `kubelet` service. `daemon_reload: true` reloads the systemd manager configuration to pick up any changes (like new service files), `state: started` ensures the service is running, and `enabled: true` ensures it starts on boot.

```
  - name: Copy hpl-benchmark.tar from host to VM
    copy:
      src: ../hpl/hpl-benchmark.tar  # path on your host machine
      dest: /tmp/hpl-benchmark.tar  # target path on the VM
      mode: '0644'
```

Copy hpl-benchmark.tar from host to VM: Uses the `copy` module to transfer a file named `hpl-benchmark.tar` from the Ansible control machine (or where the playbook is run from) to the target VM. `src` specifies the path on the host, and `dest` specifies the target path on the remote machine. `mode: '0644'` sets the file permissions.

```
  - name: Import image into containerd
    command: ctr -n=k8s.io images import /tmp/hpl-benchmark.tar
```

Import image into containerd: Uses the `command` module to execute the `ctr` command-line tool (for containerd). It imports the container image from the `/tmp/hpl-benchmark.tar` file into the `k8s.io` namespace within containerd. This makes the image available for Kubernetes to use.

Control Plane provisioning

1. **Play Definition:**

```
- name: Configure Kubernetes Control Plane
  hosts: control
  become: true
  vars:
    pod_network_cidr: "10.244.0.0/16"  # Match the CIDR for Flannel
    flannel_manifest_url: "https://github.com/flannel-
io/flannel/releases/latest/download/kube-flannel.yml"
```

   - `- name: Configure Kubernetes Control Plane` : This line provides a descriptive name for the play, indicating its purpose.
   - `hosts: control` : This specifies that this play should run only on hosts defined in the Ansible inventory that belong to the `control` group.
   - `become: true` : This indicates that Ansible should execute all tasks within this play with elevated privileges (equivalent to using `sudo`).
   - `vars:` : This section defines variables that can be used throughout this play.
2. **Variable Definitions:**

```
  vars:
    pod_network_cidr: "10.244.0.0/16"  # Match the CIDR for Flannel
```

```
    flannel_manifest_url: "https://github.com/flannel-
io/flannel/releases/latest/download/kube-flannel.yml"
```

- `pod_network_cidr: "10.244.0.0/16"` : Defines a variable `pod_network_cidr` to specify the network range for Kubernetes pods. This should match the CIDR used by the chosen network plugin (like Flannel).
- `flannel_manifest_url: "https://github.com/flannel-io/flannel/releases/latest/download/kube-flannel.yml"` : Defines a variable `flannel_manifest_url` holding the direct URL to the Flannel CNI plugin manifest file.

3. **Tasks:**

```
  - name: Initialize Kubernetes cluster with kubeadm
    command: kubeadm init --pod-network-cidr={{ pod_network_cidr }}
```

Initialize Kubernetes cluster with kubeadm: Uses the `command` module to execute the `kubeadm init` command. This command bootstraps the Kubernetes control plane on the current node. The `--pod-network-cidr` flag is set using the `pod_network_cidr` variable, which is required by the CNI plugin (Flannel in this case). The `#register: kubeadm_init_result` line is commented out but shows that the output of this command could be captured for later use.

```
  - name: Create .kube directory for vagrant user
    file:
      path: /home/vagrant/.kube
      state: directory
      owner: vagrant
      group: vagrant
      mode: '0755'
```

Create .kube directory for vagrant user: Uses the `file` module to ensure the `.kube` directory exists in the `/home/vagrant` directory. This directory is where Kubernetes configuration files (like `kubeconfig`) are typically stored. `state: directory` ensures it's a directory, and `owner` , `group` , and `mode` set the appropriate permissions for the `vagrant` user.

```
  - name: Copy admin.conf to user's .kube directory
    copy:
      src: /etc/kubernetes/admin.conf
      dest: /home/vagrant/.kube/config
      remote_src: yes
      owner: vagrant
      group: vagrant
      mode: '0644'
```

Copy admin.conf to user's .kube directory: Uses the `copy` module to copy the Kubernetes administrator configuration file (`admin.conf`) from `/etc/kubernetes/` to `/home/vagrant/.kube/config` . This allows the `vagrant` user to interact with the cluster using `kubectl` . `remote_src: yes` indicates that the source file is on the remote host. `owner` , `group` , and `mode` set permissions for the `vagrant` user.

```
  - name: Fetch the kubeconfig file to the local machine
    fetch:
      src: /home/vagrant/.kube/config
      dest: ./playground_kubeconfig.yaml
      flat: yes # This ensures the file is copied directly to the dest
path without creating the remote directory structure
```

Fetch the kubeconfig file to the local machine: Uses the `fetch` module to copy the `kubeconfig` file from the remote host (`src`) to the Ansible control machine (`dest`). `flat: yes` ensures the file is placed directly in the specified destination path without recreating the remote directory structure. This allows the user to interact with the cluster from their local machine.

```
  # ============ Install Flannel CNI Plugin ============
  - name: Download Flannel manifest
    get_url:
      url: "https://github.com/flannel-
io/flannel/releases/latest/download/kube-flannel.yml"
      dest: /tmp/kube-flannel.yml
      mode: '0644'
    register: flannel_download
```

Download Flannel manifest: Uses the `get_url` module to download the Flannel CNI manifest file from the URL specified by the `flannel_manifest_url` variable and save it to `/tmp/kube-flannel.yml` on the remote host. `mode: '0644'` sets the file permissions. The `register: flannel_download` captures the result of this task for use in subsequent tasks.

```
  - name: Install Flannel network plugin
    shell: kubectl apply -f /tmp/kube-flannel.yml
    become: false  # Run as regular user, not root
    environment:
      KUBECONFIG: /home/vagrant/.kube/config
    register: flannel_install_result
    when: flannel_download is succeeded
```

Install Flannel network plugin: Uses the `shell` module to apply the downloaded Flannel manifest using `kubectl apply -f` . This installs the Flannel CNI plugin, enabling pod networking. `become: false` ensures the command runs as the default user (vagrant),

not root. The `environment` directive sets the `KUBECONFIG` environment variable, telling `kubectl` where to find the configuration file. `register: flannel_install_result` captures the output, and `when: flannel_download is succeeded` ensures this task only runs if the download task was successful.

```
  - name: Show Flannel installation result
    debug:
      var: flannel_install_result.stdout_lines
    when: flannel_install_result is defined
```

Show Flannel installation result: Uses the `debug` module to print the standard output lines from the `flannel_install_result` variable. This is useful for verifying the output of the Flannel installation command. The `when: flannel_install_result is defined` condition ensures this task only runs if the previous task successfully registered a result.

```
  # =========== Generate Join Command for Workers ===========
  - name: Generate node join command
    command: kubeadm token create --print-join-command
    register: join_command
```

Generate node join command: Uses the `command` module to execute `kubeadm token create --print-join-command`. This command generates a new join token and prints the full `kubeadm join` command that can be used to add worker nodes to the cluster. `register: join_command` captures the output.

```
  - name: Copy join command to local file
    local_action: copy content="{{ join_command.stdout_lines[0] }}"
dest="./join-command"
    become: false  # Explicitly disable privilege escalation for local
action
```

Copy join command to local file: Uses `local_action` with the `copy` module to copy the generated join command to a file named `join-command` on the Ansible control machine (where the playbook is being run). `content="{{ join_command.stdout_lines[0] }}"` takes the first line of the standard output from the `join_command` task as the content for the file. `dest="./join-command"` specifies the destination path on the local machine. `become: false` is explicitly set for local actions as they run on the control machine.

Worker Node provisioning

Here is an analysis of the provided Ansible playbook snippet for configuring Kubernetes worker nodes.

1. **Play Definition:**

```
- name: Configure Kubernetes Worker Nodes
  hosts: workers
  become: true
```

- `- name: Configure Kubernetes Worker Nodes` : This line provides a descriptive name for the play, indicating its purpose.
- `hosts: workers` : This specifies that this play should run only on hosts defined in the Ansible inventory that belong to the `workers` group.
- `become: true` : This indicates that Ansible should execute all tasks within this play with elevated privileges (equivalent to using `sudo`).

2. **Tasks:**

```
  - name: Copy join command to worker nodes
    copy:
      src: join-command
      dest: /tmp/join-command.sh
      mode: 0777
```

Copy join command to worker nodes: Uses the `copy` module to transfer the `join-command` file (which was fetched from the control plane node to the Ansible control machine in the previous play) to the `/tmp/join-command.sh` path on each worker node. `src` refers to the file on the Ansible control machine, and `dest` is the target path on the remote worker node. `mode: 0777` sets the file permissions to be executable by all users, which is necessary for the next step.

```
  - name: Join worker nodes to the cluster
    command: sh /tmp/join-command.sh
    register: join_result
```

Join worker nodes to the cluster: Uses the `command` module to execute the shell script located at `/tmp/join-command.sh` on each worker node. This script contains the `kubeadm join` command generated by the control plane play, which connects the worker node to the Kubernetes cluster. `register: join_result` captures the standard output and standard error of this command for potential debugging or verification.

```
  - name: Display join result
    debug:
      var: join_result.stdout_lines
```

Display join result: Uses the `debug` module to print the standard output lines captured in the `join_result` variable from the previous task. This allows you to see the output of the `kubeadm join` command, which can be helpful for confirming a successful join or troubleshooting issues.

```
    # Optional: Clean up the join command file
    - name: Remove join command file
      file:
        path: /tmp/join-command.sh
        state: absent
```

Remove join command file: Uses the `file` module to remove the temporary `join-command.sh` file from the `/tmp/` directory on the worker nodes. `state: absent` ensures that the file is deleted if it exists. This is an optional cleanup step.

Docker image

The same `hpl-test` Docker image used in the container section is utilized here. Please refer to that section for further details.

Hpl pod manifest

```
apiVersion: v1
kind: Namespace
metadata:
  name: hpl-benchmarks
```

Namespace: Creates a dedicated space (`hpl-benchmarks`) to isolate and organize the benchmark resources.

```
apiVersion: v1
kind: Pod
metadata:
  name: hpl-benchmark
  namespace: hpl-benchmarks
```

Pod: Defines the benchmark execution unit `hpl-benchmark` within that isolated namespace.

```
spec:
  restartPolicy: Never
  containers:
    - name: hpl
      image: hpl-benchmark:latest
      imagePullPolicy: Never
```

- `restartPolicy: Never` : Ensures the Pod runs once and does not restart after completion or failure, suitable for a benchmark job.
- `image: hpl-benchmark:latest` : Specifies the container image containing the benchmark code.

- `imagePullPolicy: Never` : Assumes the image is already available locally on the node.

```
    resources:
      requests:
        cpu: "3"
        memory: "3Gi"
      limits:
        cpu: "3"
        memory: "3Gi"
```

Resources: Allocates precisely 3 CPU cores and 3Gi of memory for the benchmark container.

```
    volumeMounts:
      - name: hpl-config-volume
        mountPath: /etc/hpl-config
```

Configuration Volume Mount: Mounts a volume (sourced from a `ConfigMap`) at `/etc/hpl-config` inside the container to provide the benchmark's configuration file (`HPL.dat`).

```
    command: ["/bin/bash", "-c"]
    args:
       - |
        cp /etc/hpl-config/HPL.dat .;
        mpirun -np 3 ./xhpl
```

Execution Command: Runs a bash script that first copies the `HPL.dat` config to the current directory, then executes the HPL benchmark `./xhpl` using `mpirun` with 3 parallel processes.

```
  volumes:
    - name: hpl-config-volume
      configMap:
        name: hpl-config
```

Volume Definition: Defines the `hpl-config-volume` source as a `ConfigMap` named `hpl-config` , making its content available to the container

Setup

Build the docker image for the hpl test.

```
> cd hpl
> docker build -t hpl-benchmark .
```

Save the docker image

```
> docker save -o hpl-benchmark.tar hpl-benchmark:latest
```

Start the Vagrant environment:

```
> vagrant up --provider=libvirt --no-parallel
```

Accessing the Kubernetes Cluster

Once the vagrant up command completes successfully, you can interact with your Kubernetes cluster using `kubectl`.

For using `kubectl` without accessing the control node set the `KUBECONFIG` environment variable in your host.

```
export KUBECONFIG=$(pwd)/kubernetes-setup/playground_kubeconfig.yaml
```

Verify the cluster status

You should be able to see the status of your nodes.

```
> kubectl get nodes
NAME          STATUS   ROLES           AGE     VERSION
kube-01       Ready    <none>          3m22s   v1.32.4
kube-02       Ready    <none>          72s     v1.32.4
kube-master   Ready    control-plane   5m33s   v1.32.4
```

Install Kube-Prometheus-stack

First, register the chart's repository in your Helm client:

```
> helm repo add prometheus-community https://prometheus-
community.github.io/helm-charts
"prometheus-community" has been added to your repositories
```

Next, update your repository lists to discover the chart:

```
> helm repo update
```

Now you can run the following command to deploy the chart into a new namespace in your cluster:

```
> helm install kube-prometheus-stack \
  --create-namespace \
  --namespace kube-prometheus-stack \
  prometheus-community/kube-prometheus-stack
```

It can take a couple of minutes for the chart's components to start. Run the following command to check how they're progressing:

```
kubectl -n kube-prometheus-stack get pods
NAME                                                     READY   STATUS
RESTARTS      AGE
alertmanager-kube-prometheus-stack-alertmanager-0        2/2     Running
1 (66s ago)   83s
kube-prometheus-stack-grafana-5cd658f9b4-cln2c           3/3     Running
0             99s
kube-prometheus-stack-kube-state-metrics-b64cf5876-52j8l 1/1     Running
0             99s
kube-prometheus-stack-operator-754ff78899-669k6          1/1     Running
0             99s
kube-prometheus-stack-prometheus-node-exporter-vdgrg     1/1     Running
0             99s
prometheus-kube-prometheus-stack-prometheus-0            2/2     Running
0             83s
```

Once all the Pods show as Running, your monitoring stack is ready to use. The data exposed by the exporters will be automatically scraped by Prometheus

Open Grafana dashboards

Start a new Kubectl port forwarding session to access the Grafana UI. Use port 80 as the target because this is what the Grafana service binds to.

You can map it to a different local port, such as 8080, in this example:

```
kubectl port-forward -n kube-prometheus-stack svc/kube-prometheus-stack-
grafana 8080:80
Forwarding from 127.0.0.1:8080 → 3000
Forwarding from [::1]:8080 → 3000
```

Next visit `http://localhost:8080` in your browser. You'll see the Grafana login page. The default user account is `admin` with a password of `prom-operator`.

Viewing the resource consumption of individual pods with "Kubernetes / Compute Resources / Pod.

Run HPL test

the hpl test is deployed with the `hpl-single.yaml` manifest. before running the test you must set the `HPL.dat` file in `/hpl` and create the configmap.

Create Config map

```
kubectl create configmap hpl-config --from-file=./hpl/HPL.dat -n hpl-
benchmarks
```

Check Config map

```
kubectl get configmaps -n hpl-benchmarks
```

Delete config map

```
kubectl delete configmap hpl-config -n hpl-benchmarks
```

Run hpl test on a single pod

```
kubectl apply -f hpl/hpl-single.yaml
```

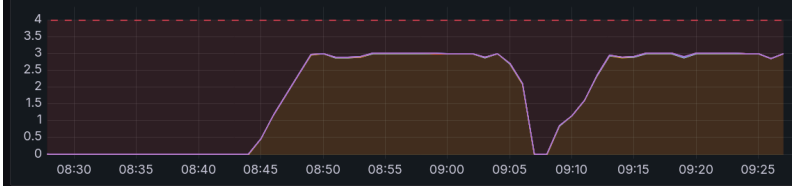
Run a standalone hpl test on each node

```
kubectl apply -f hpl/hpl-double.yaml
```

Monitoring on Grafana

I executed the HPL benchmark, gradually increasing the problem size until it exhausted the available memory within the pod, resulting in its termination by Kubernetes.

CPU Usage



Name	Last *
max capacity	4
hpl-benchmark	3.00
kube-flannel-ds-st4fv	0.00418
kube-prometheus-stack-grafana-5bdf779668-xqb24	0.00457
kube-prometheus-stack-kube-state-metrics-7976475bf5-2vhsj	0.000990
kube-prometheus-stack-prometheus-node-exporter-zrphb	0.000678

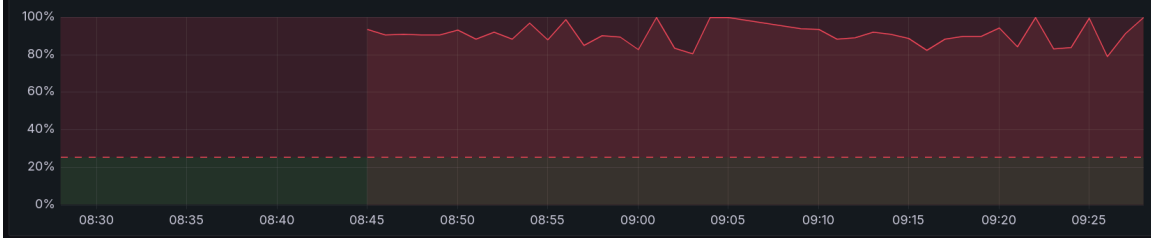
CPU Quota

Pod	CPU Usage	CPU Requests	CPU Requests %	CPU Limits	CPU Limits %
kube-prometheus-sta...	0.00980				
kube-prometheus-sta...	0.00103				
kube-flannel-ds-st4fv	0.00433	0.100	4.33%		
hpl-benchmark	2.90	3	96.8%	3	96.8%

CPU Usage

Name	Last *
hpl	2.90
requests	3
limits	3

CPU Throttling

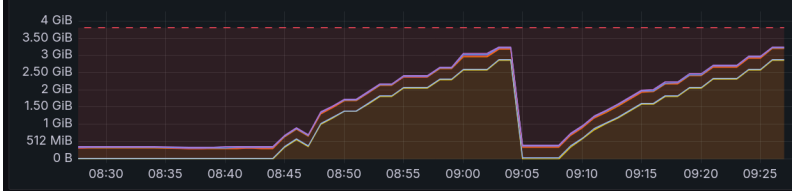


Name	Last *
hpl	100%

CPU Quota

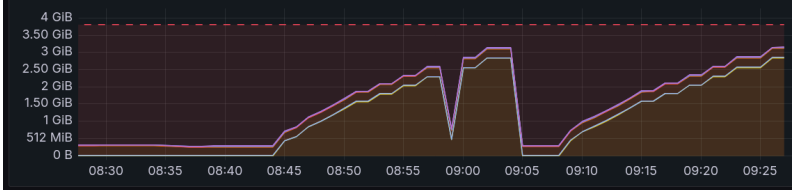
Container	CPU Usage	CPU Requests	CPU Requests %	CPU Limits	CPU Limits %
hpl	2.90	3	96.8%	3	96.8%

Memory Usage (w/cache)



max capacity	3.83 GiB
hpl-benchmark	2.86 GiB
kube-flannel-ds-st4fv	19.9 MiB
kube-prometheus-stack-grafana-5bdf779668-xqb24	325 MiB
kube-prometheus-stack-kube-state-metrics-7976475bf5-2vhsj	22.1 MiB
kube-prometheus-stack-prometheus-node-exporter-zrphb	13.3 MiB
kube-proxy-jgnv2	22.9 MiB

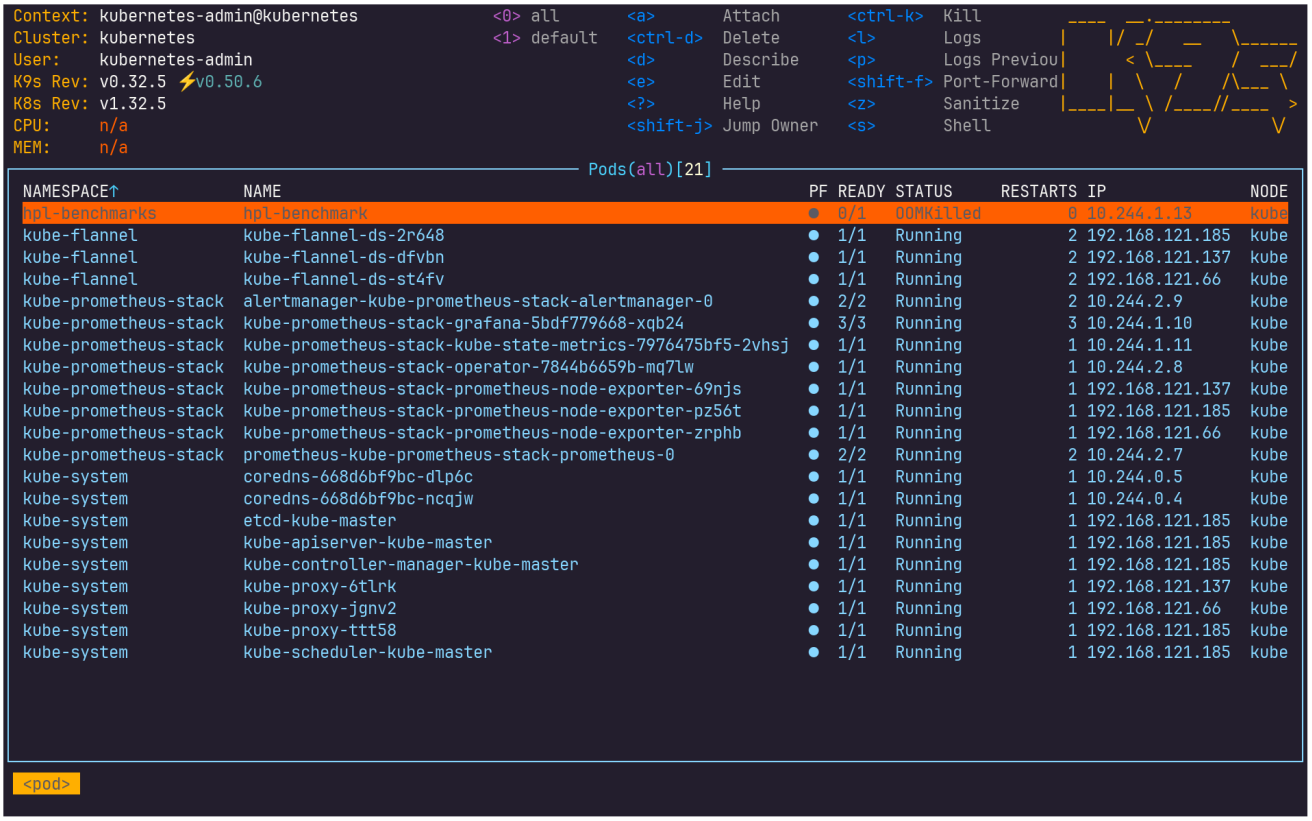
Memory Usage (w/o cache)

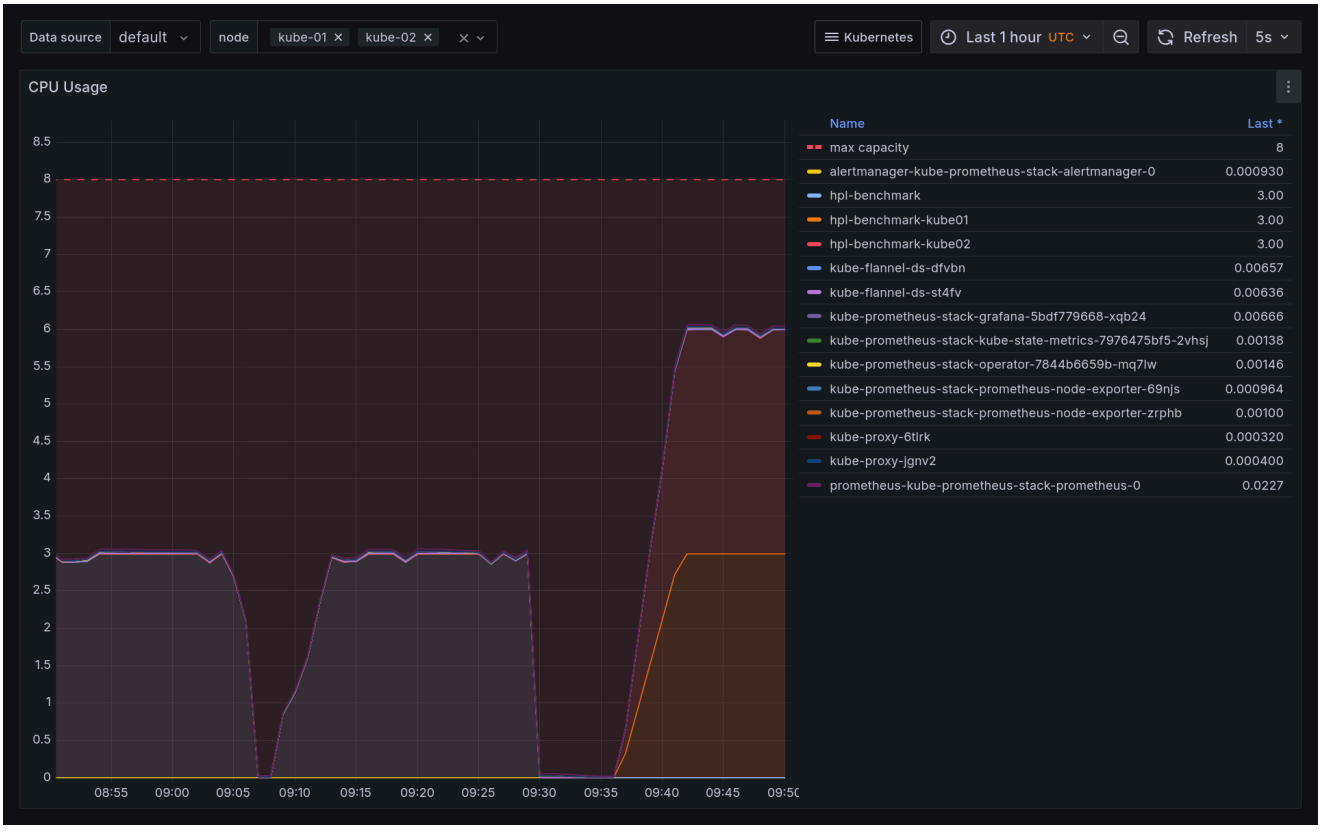


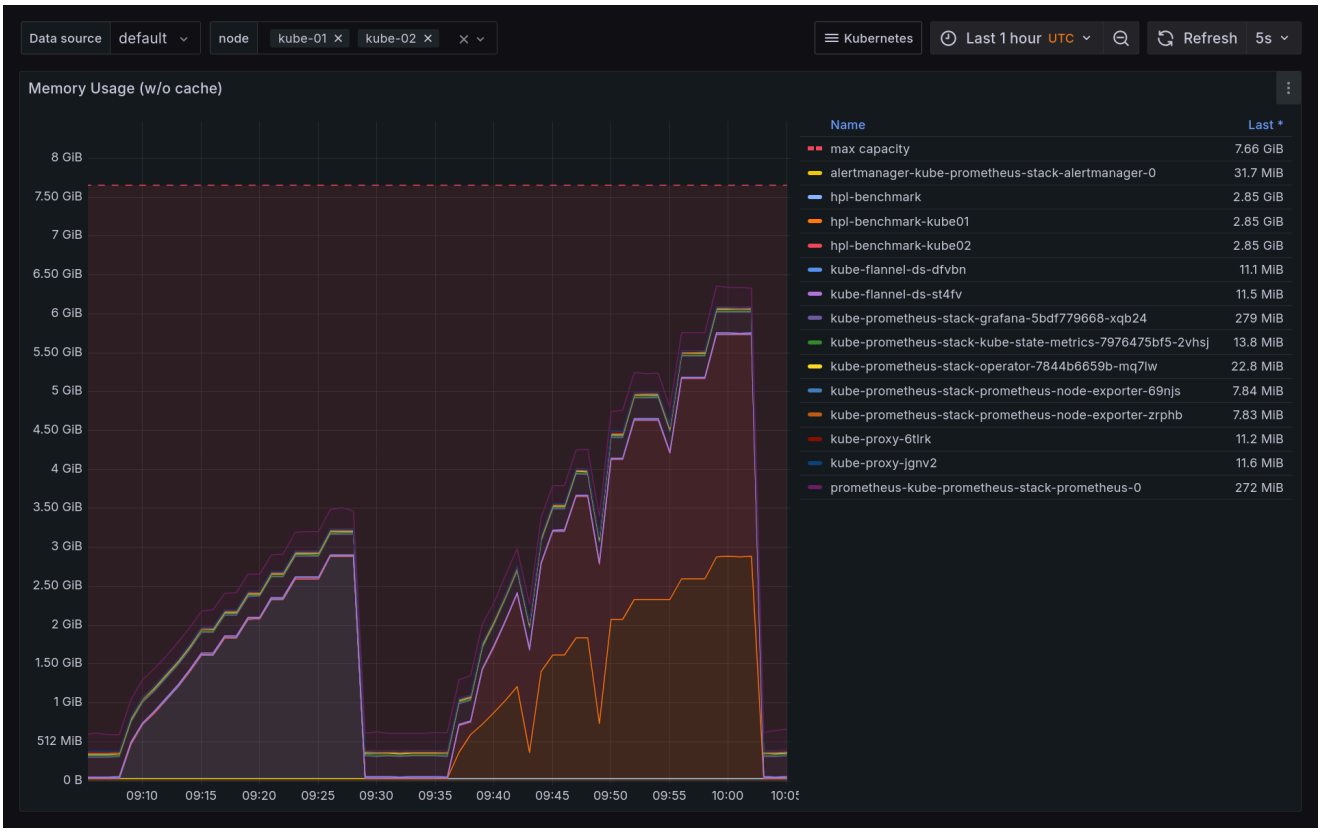
Name	Last *
max capacity	3.83 GiB
hpl-benchmark	2.85 GiB
kube-flannel-ds-st4fv	11.6 MiB
kube-prometheus-stack-grafana-5bdf779668-xqb24	271 MiB
kube-prometheus-stack-kube-state-metrics-7976475bf5-2vhsj	13.5 MiB
kube-prometheus-stack-prometheus-node-exporter-zrphb	7.65 MiB

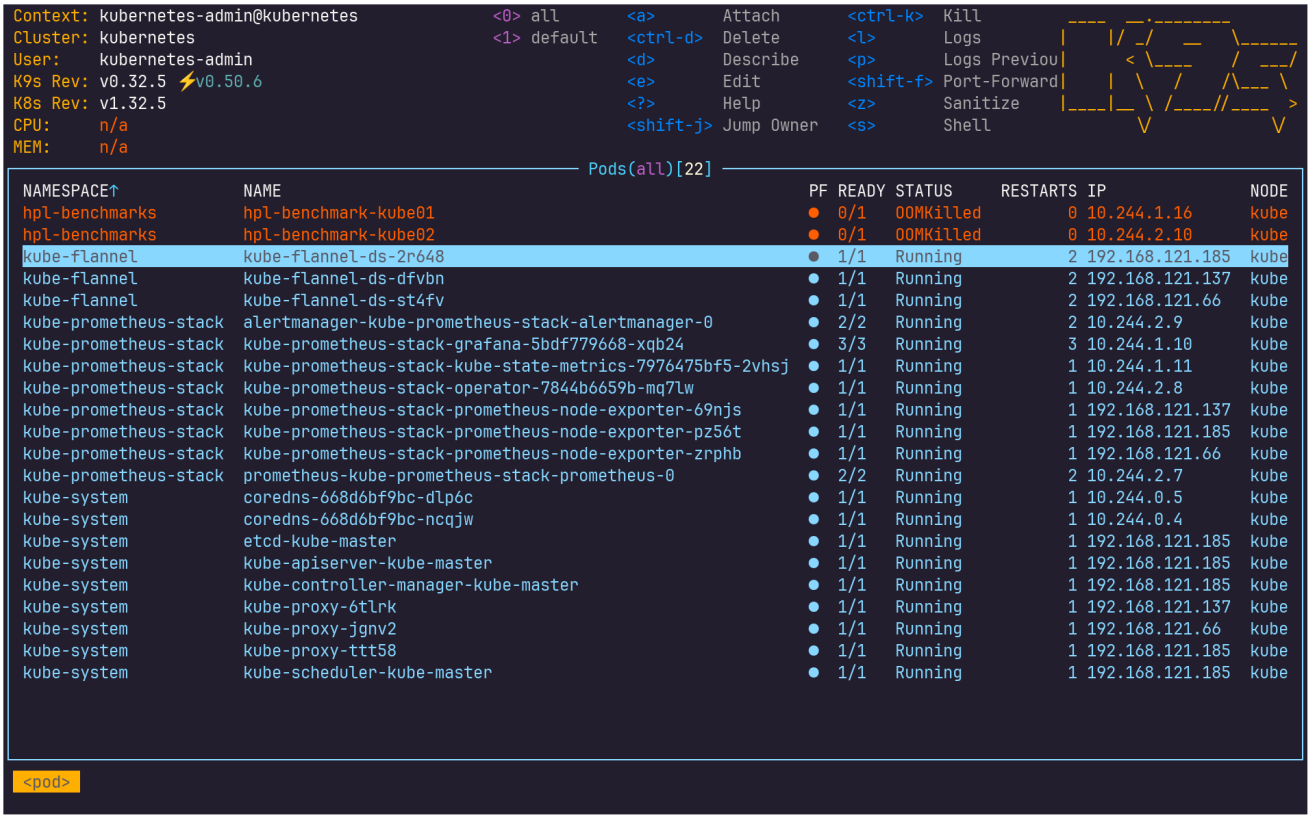
Memory Quota

Pod	Memory Usage	Memory Requests	Memory Requests %	Memory Limits	Memory Limits %	Memory Usage (RSS	Memory Usage (Cac
kube-proxy-jg...	22.9 MiB					11.6 MiB	14.9 MiB
kube-prometh...	21.9 MiB					13.5 MiB	13.4 MiB
kube-prometh...	13.3 MiB					7.65 MiB	5.88 MiB
hpl-benchmark	2.86 GiB	3 GiB	95.4%	3 GiB	95.4%	2.85 GiB	9.52 MiB


Context: kubernetes-admin@kubernetes
Cluster: kubernetes
User: kubernetes-admin
K9s Rev: v0.32.5 ⚡v0.50.6
K8s Rev: v1.32.5
CPU: n/a
MEM: n/a
<0> all
<1> default
<a> Attach
<ctrl-d> Delete
<d> Describe
<e> Edit
<?> Help
<shift-j> Jump Owner
<ctrl-k> Kill
<l> Logs
<p> Logs Previou
<shift-f> Port-Forward
<z> Sanitize
<s> Shell
Pods(all)[21]
NAMESPACE↑ NAME PF READY STATUS RESTARTS IP NODE
hpl-benchmarks hpl-benchmark 0/1 OOMKilled 0 10.244.1.13 kube
kube-flannel kube-flannel-ds-2r648 1/1 Running 2 192.168.121.185 kube
kube-flannel kube-flannel-ds-dfvbn 1/1 Running 2 192.168.121.137 kube
kube-flannel kube-flannel-ds-st4fv 1/1 Running 2 192.168.121.66 kube
kube-prometheus-stack alertmanager-kube-prometheus-stack-alertmanager-0 2/2 Running 2 10.244.2.9 kube
kube-prometheus-stack kube-prometheus-stack-grafana-5bdf779668-xqb24 3/3 Running 3 10.244.1.10 kube
kube-prometheus-stack kube-prometheus-stack-kube-state-metrics-7976475bf5-2vhsj 1/1 Running 1 10.244.1.11 kube
kube-prometheus-stack kube-prometheus-stack-operator-7844b6659b-mq7lw 1/1 Running 1 10.244.2.8 kube
kube-prometheus-stack kube-prometheus-stack-prometheus-node-exporter-69njs 1/1 Running 1 192.168.121.137 kube
kube-prometheus-stack kube-prometheus-stack-prometheus-node-exporter-pz56t 1/1 Running 1 192.168.121.185 kube
kube-prometheus-stack kube-prometheus-stack-prometheus-node-exporter-zrphb 1/1 Running 1 192.168.121.66 kube
kube-prometheus-stack prometheus-kube-prometheus-stack-prometheus-0 2/2 Running 2 10.244.2.7 kube
kube-system coredns-668d6bf9bc-dlp6c 1/1 Running 1 10.244.0.5 kube
kube-system coredns-668d6bf9bc-ncqjw 1/1 Running 1 10.244.0.4 kube
kube-system etcd-kube-master 1/1 Running 1 192.168.121.185 kube
kube-system kube-apiserver-kube-master 1/1 Running 1 192.168.121.185 kube
kube-system kube-controller-manager-kube-master 1/1 Running 1 192.168.121.185 kube
kube-system kube-proxy-6tlrk 1/1 Running 1 192.168.121.137 kube
kube-system kube-proxy-jgnv2 1/1 Running 1 192.168.121.66 kube
kube-system kube-proxy-ttt58 1/1 Running 1 192.168.121.185 kube
kube-system kube-scheduler-kube-master 1/1 Running 1 192.168.121.185 kube
<pod>


Data source default
node kube-01 kube-02
Kubernetes
Last 1 hour UTC
Refresh 5s
CPU Usage
8.5
8
7.5
7
6.5
6
5.5
5
4.5
4
3.5
3
2.5
2
1.5
1
0.5
0
08:55 09:00 09:05 09:10 09:15 09:20 09:25 09:30 09:35 09:40 09:45
Name Last *
max capacity 8
alertmanager-kube-prometheus-stack-alertmanager-0 0.000930
hpl-benchmark 3.00
hpl-benchmark-kube01 3.00
hpl-benchmark-kube02 3.00
kube-flannel-ds-dfvbn 0.00657
kube-flannel-ds-st4fv 0.00636
kube-prometheus-stack-grafana-5bdf779668-xqb24 0.00666
kube-prometheus-stack-kube-state-metrics-7976475bf5-2vhsj 0.00138
kube-prometheus-stack-operator-7844b6659b-mq7lw 0.00146
kube-prometheus-stack-prometheus-node-exporter-69njs 0.000964
kube-prometheus-stack-prometheus-node-exporter-zrphb 0.00100
kube-proxy-6tlrk 0.000320
kube-proxy-jgnv2 0.000400
prometheus-kube-prometheus-stack-prometheus-0 0.0227





Troubleshooting

Pending status indefinitely

```
> kubectl get pods -n hpl-benchmarks
NAME             READY   STATUS      RESTARTS   AGE
hpl-benchmark    0/1     Pending   0          52s
> kubectl describe pod hpl-benchmark -n hpl-benchmarks
```

```
...
Events:
  Type     Reason       Age                 From               Message
  ----     ------       ----                ----               -------
  Normal   Scheduled    107s                default-scheduler
Successfully assigned hpl-benchmarks/hpl-benchmark to kube-01
  Warning  FailedMount  43s (x8 over 107s)  kubelet
MountVolume.SetUp failed for volume "hpl-config-volume" : configmap "hpl-
config" not found
```

The pod `hpl-benchmark` in the `hpl-benchmarks` namespace is failing to start due to a `FailedMount` error. This means that the pod is configured to mount a volume named `hpl-config-volume`, which relies on a Kubernetes ConfigMap named `hpl-config`. However, the Kubernetes cluster cannot find a ConfigMap with that name in the `hpl-benchmarks` namespace.

To fix this issue, you need to create the `hpl-config` ConfigMap in the `hpl-benchmarks` namespace.

```
> kubectl describe pod hpl-benchmark -n hpl-benchmarks
...
Events:
  Type     Reason            Age   From               Message
  ----     ------            ----  ----               -------
  Warning  FailedScheduling  47s   default-scheduler  0/3 nodes are
available: 1 node(s) had untolerated taint {node-
role.kubernetes.io/control-plane: }, 2 Insufficient cpu. preemption: 0/3
nodes are available: 1 Preemption is not helpful for scheduling, 2 No
preemption victims found for incoming pod.
```

Based on the `Events` section of the `kubectl describe pod` output for `hpl-benchmark`, the pod failed to schedule due to two primary reasons across the three available nodes:

1. **Untolerated Taint:** One node in the cluster has a taint `node-role.kubernetes.io/control-plane`. This is the control plane nodes. The `hpl-benchmark` pod does not have this toleration, making that node unavailable for scheduling.
2. **Insufficient CPU:** The other two nodes in the cluster do not have enough available CPU resources to meet the requirements of the `hpl-benchmark` pod.

Image error

```
> kubectl get pods -n hpl-benchmarks
NAME            READY   STATUS             RESTARTS   AGE
hpl-benchmark   0/1     ErrImageNeverPull  0          95s
> kubectl describe pod hpl-benchmark -n hpl-benchmarks
```

```
...
Events:
  Type     Reason             Age                From               Message
  ----     ------             ----               ----               -------
  Normal   Scheduled          21s                default-scheduler
Successfully assigned hpl-benchmarks/hpl-benchmark to kube-01
  Warning  ErrImageNeverPull  8s (x3 over 21s)   kubelet
Container image "hpl-benchmark:latest" is not present with pull policy of
Never
  Warning  Failed             8s (x3 over 21s)   kubelet            Error:
ErrImageNeverPull
```

The error `ErrImageNeverPull` indicates that the container image required by the pod `hpl-benchmark:latest` was not found on the node where the pod was scheduled, and the pod's container definition has the `imagePullPolicy` explicitly set to `Never`. Because Kubernetes is instructed *not* to pull the image from a registry when this policy is set and the image is not locally available, the container creation fails.

```
> kubectl get pods -n hpl-benchmarks
NAME            READY   STATUS   RESTARTS   AGE
hpl-benchmark   0/1     Error    0          17s
> kubectl logs hpl-benchmark -n hpl-benchmarks
--------------------------------------------------------------------------
There are not enough slots available in the system to satisfy the 16
slots that were requested by the application:

  ./xhpl
...
```

Analysis of the pod logs indicates that the HPL benchmark failed because it attempted to allocate 16 processing slots (likely corresponding to cores or threads), but the system did not have sufficient resources available to fulfill this request. The pod entered an `Error` state as a result

OOMKilled

```
> kubectl get pods -n hpl-benchmarks
NAME            READY   STATUS      RESTARTS   AGE
hpl-benchmark   0/1     OOMKilled   0          52s
```

The `OOMKilled` status stands for **Out Of Memory Killed**. This means that the Kubernetes node where your pod was running detected that the pod was consuming more memory than it was allowed (either exceeding its defined memory limit or, if no limit was set, consuming excessive memory that threatened the node's stability).