Kubernetes Dynamic Volume Provisioning Benchmark

A Project Report

submitted in partial fulfillment of the requirements

for the award of the degree of

Masters

in

COMPUTER SCIENCE

by

MS. SAYALI BHONGADE

MR. VISHAL KUMAR BHAGCHANDANI

Under the guidance of

MENG TANG

DEPARTMENT OF COMPUTER SCIENCE

ILLINOIS INSTITUTE OF TECHNOLOGY

CHIAGO, 60616

CERTIFICATE

The report entitled **Kubernetes Dynamic Volume Provisioning Benchmark** submitted by Sayali Bhongade (A2054430), Vishal Kumar Bhagchandani (A20526905) is approved for the partial fulfilment of the requirement for the award of master's in computer science and project for cloud computing.

Dr. **Xian-He Sun** Guide
former Department Chairman of the Department of Computer Science
&
Ms. Meng Tang project guide
PhD in Computer Science

Examiners:

1. _____ (Name:_____)
2. _____ (Name:_____)

Place : Illinois Institute of Technology
Date:

Acknowledgement

Words are inadequate to express the overwhelming sense of gratitude and humble regards to Meng Tang for constant motivation, support, expert guidance, constant and constructive supervision and constructive suggestion for the submission of our project report on "**Kubernetes Dynamic Volume Provisioning Benchmark".**

We express our gratitude to the professor Dr. **Xian-He Sun** Department of Computer Science for invaluable suggestions and constant encouragement all through the seminar work.

Index

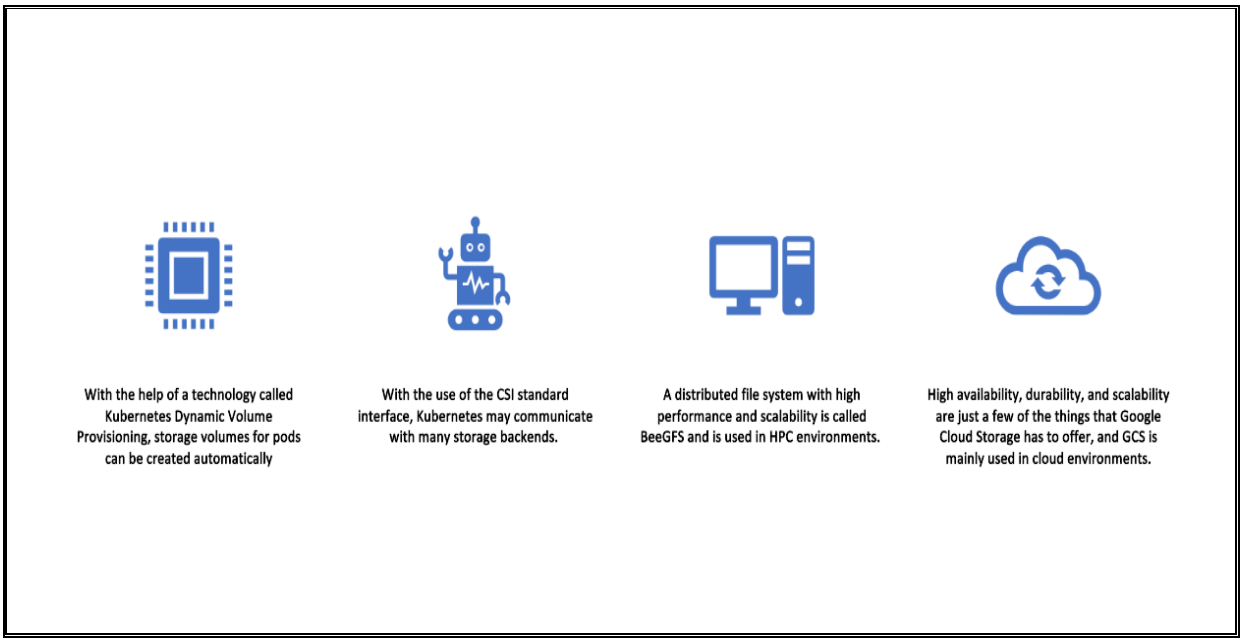
Abstract

Container technology has been expanding quickly and is being widely embraced in cloud computing because it allows for the creation of individualized, adaptable, and separated environments for executing programmers. The introduction of container orchestration solutions like Kubernetes has considerably increased the ability to dynamically provision containers to run large-scale applications. However, minimal study has been done on Kubernetes' storage performance. In this project, BeeGFS will be used as benchmarks to measure and analyze the performance of Kubernetes Dynamic Volume Provisioning for Parallel file systems. We will also compare the performance against the default Linux file system. This project aims to show the performance effect of containerized and provisioned PFS, which was not well-studied before. This project seeks to demonstrate a practical illustration of containerization and storage provisioning for deploying PFS (Parallel File System) using established tools such as Kubernetes. Additionally, the project aims to conduct a performance analysis and comparison among baseline scenarios (utilizing a Linux File System PFS. By delving into the performance impact of containerized and provisioned PFS, this endeavor addresses an area that has not been extensively explored in previous studies.

Background Information

Leveraging Kubernetes Dynamic Volume Provisioning, storage volumes for pods can be automatically generated, facilitated by a Container Storage Interface (CSI) driver. The CSI standard interface allows Kubernetes to interface with various storage backends. BeeGFS, recognized for its high performance and scalability, finds utility in High-Performance Computing (HPC) environments, frequently serving big data applications and scientific computing needs. On the other hand, Google Cloud Storage (GCS) stands out for its high availability, durability, and scalability, making it a preferred choice in cloud environments for storing container images and other Kubernetes artifacts. It's noteworthy that BeeGFS is prominent in the HPC ecosystem, whereas Google Cloud Storage is widely utilized in cloud environments. This distinction sets the stage for exploring both filesystems, emphasizing the diverse contexts in which they are employed.

Diagrammatic representation Information:



Background Technologies and System Requirements

- Docker: Containerization technology.
- Kubernetes: Container orchestration solution (version: 1.28).
- BeeGFS: High-performance distributed file system(version: 7.4.0).
- Chameleon Cloud To deploy the solution.
- AWS (Amazon Web Services): an alternat solution to chameleon cloud.
- Open MPI: version 4.1.3
- IOR: version 2.17.9

- **Ubuntu System Specification :**
- **Operating System:**
- Ubuntu (Specify the version, e.g., Ubuntu 22.04 LTS)
- **Hardware Requirements:**
- Processor: Multi-core processor (e.g., Intel Core i5 or equivalent)
- RAM: Minimum 16 GB (32 GB or more recommended for optimal performance)
- Storage: Minimum 100 GB available disk space (SSD recommended for better performance)

1. Introduction

1.1 Project Name: "Kubernetes Dynamic Volume Provisioning Benchmark"

1.2 Goal : This project explores Kubernetes storage performance, focusing on Volumes and Container Storage Interface (CSI). Students learn to use various CSI options for Kubernetes Volumes and conduct benchmarking of the Dynamic Volume Provisioning feature. The comparison includes two storage backends, BeeGFS, both compatible with Kubernetes CSI. The goal is to deepen students' knowledge of Kubernetes Volumes and enhance their skills in benchmarking and analyzing storage performance. The project introduces practical insights by comparing storage solutions commonly employed in different scenarios.

1.3 Description: The main purpose of the project is to investigate the storage performance of Kubernetes, a widely adopted container technology in cloud computing. It focuses on Kubernetes Volumes, teaching students to use different Container Storage Interface (CSI) options for volume management. The project emphasizes benchmarking and analyzing the performance of Kubernetes' Dynamic Volume Provisioning. Two storage backends, BeeGFS, both with available CSI for Kubernetes, will be compared against the default Linux file system (e.g., XFS, BTRFS, EXT4). The goal is to deepen students' understanding of Kubernetes Volumes and enhance their skills in benchmarking and performance analysis. The project introduces a practical element by comparing storage solutions commonly used in diverse contexts.

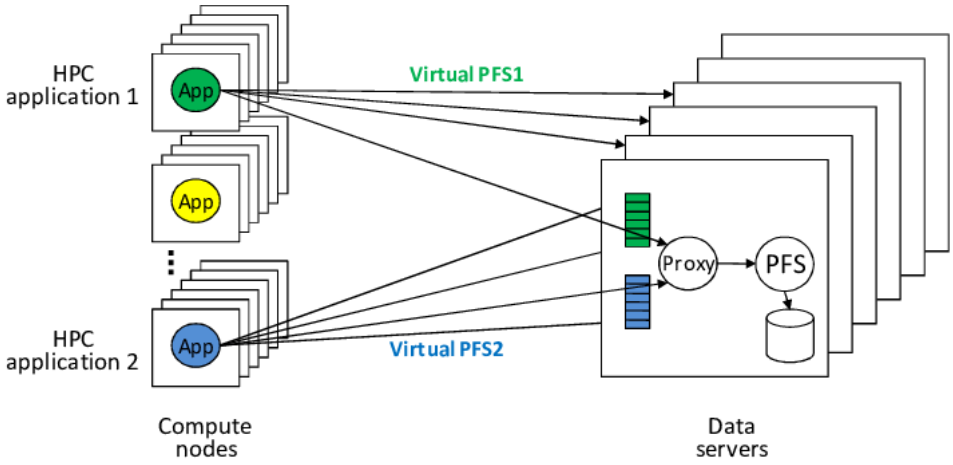
2. Problem Statement:

The behavior of the HPC cluster w.r.t Kubernetes has not been extensively researched and emphasis on dynamic volume provisioning for PFS is void as of now. Identifying a suitable storage backend for a particular application may therefore be difficult. It is vital to benchmark and assess how Kubernetes Dynamic Volume Provisioning functions on multiple storage backends so that users can select the appropriate storage backend.

2.1 Problem in Existing Research:

The statement highlights a significant gap in research on the storage performance of Kubernetes, posing challenges for users in selecting suitable storage backends for their applications. Due to this lack of extensive research, users may find it difficult to identify the most appropriate storage solutions, emphasizing the need for benchmarking to understand how Kubernetes Dynamic Volume Provisioning functions across diverse backends. The absence of comprehensive guidance on storage backends underscores the importance of research studies and documentation to assist

users in making informed decisions. This problem affects users' ability to optimize storage configurations, potentially impacting the performance and reliability of applications running on Kubernetes. Addressing this issue requires increased research efforts, benchmarking studies, and the development of best practices tailored to Kubernetes storage performance. Ultimately, a more thorough understanding of how Kubernetes interacts with different storage solutions is crucial for enhancing the overall efficiency of containerized applications.



Few research has been done, below attached is the links of the documents:

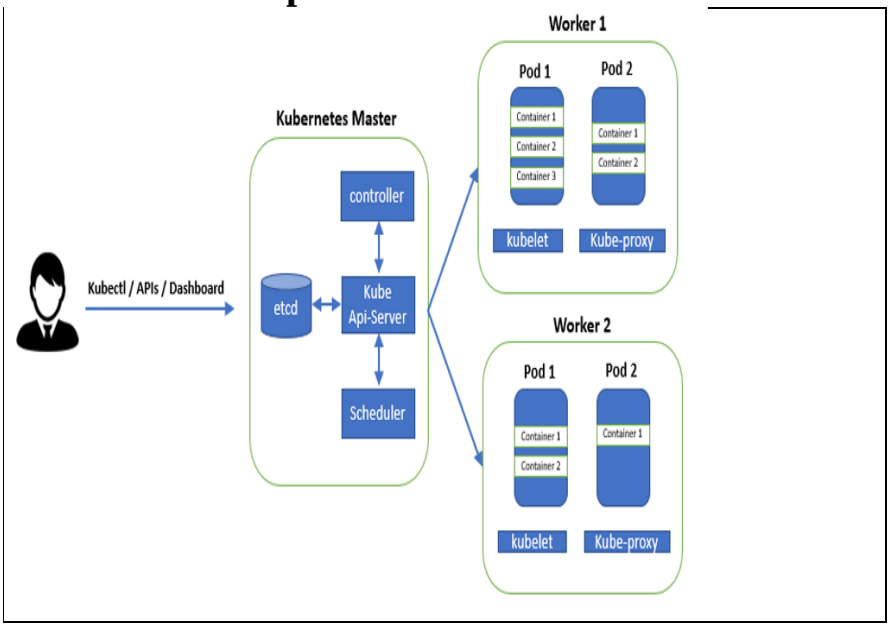
- Performance Analysis Of An Hyperconverged Infrastructure Using Docker Containers And Glusterfs
- Rodrigo Leite and Priscila Solis. "Performance analysis of data storage in a hyperconverged
- infrastructure using docker and GlusterFS." 2019 XLV Latin American Computing Conference (CLEI). IEEE, 2019. [Article]
- Jay Lofstead, Joshua Baker, and Andrew Younge. "Data pallets: containerizing storage for
- reproducibility and traceability." International Conference on High Performance Computing. Springer, Cham, 2019. [Article]
- Izzet Yildirim, Meng Tang, Anthony Kougkas, and Xian-He Sun. "Performance Analysis of
- Containerized OrangeFS in HPC Environment." SC'21. [Poster][Extended Abstract]
- Volumes: https://kubernetes.io/docs/concepts/storage/volumes/
- CSI: https://kubernetes-csi.github.io/docs/introduction.html

3. Background Technologies Used:

3.1 Kubernetes:

- Kubernetes is an open-source container orchestration platform .
- Kubernetes simplifies container deployment and management, ensuring applications run consistently across various environments.
- Automated load balancing distributes traffic efficiently, while automated rollouts and rollbacks enable seamless updates.

3.1.1 Kubernetes Setup:



3.1.2 Kubernetes Cluster Setup:

Components: Master

- and Worker Nodes
- Key Components: etcd, kube-apiserver, kube-controller-manager, kube-scheduler, kubelet, kube-proxy

Installation Method:

- Choose between kubeadm, kops, or managed Kubernetes services (e.g., GKE, AKS, EKS)

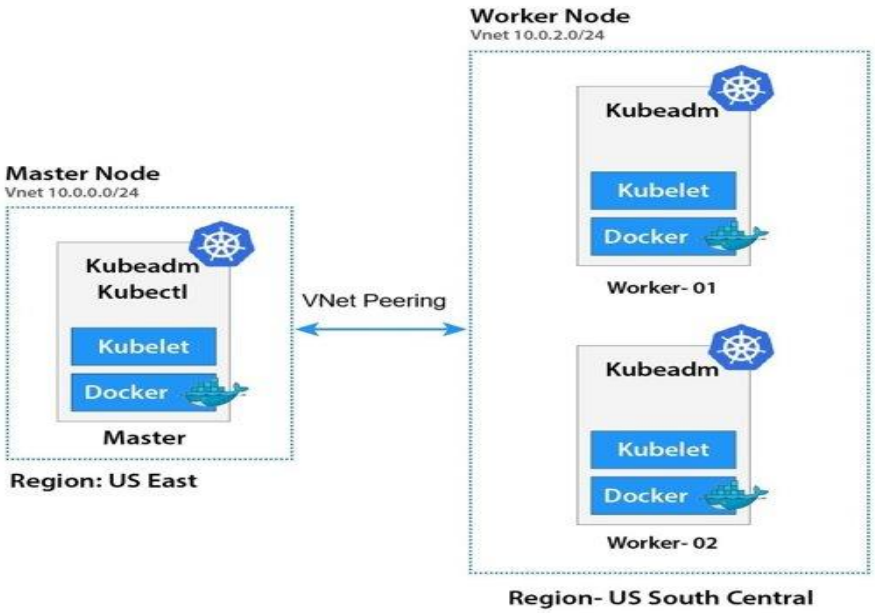
Prerequisites:

- Container Runtime: Docker, containers

- Networking: Choose a suitable CNI (Container Network Interface) plugin

Networking Setup:

- Configuring Pod-to-Pod communication and select CNI.



3.1.3 Installation steps:

Prerequisites:

1. Hardware Requirements:

Ensure that each machine in your cluster meets the minimum hardware requirements.

Allocate enough resources such as CPU, memory, and storage.

2. Operating System:

Choose a Linux distribution supported by Kubernetes (e.g., Ubuntu, CentOS, or RHEL).

Update the system and install necessary packages like `docker` and `kubelet`.

1. Disable Swap:

Kubernetes requires swap to be disabled on all nodes.

2. Install Docker:

Install the Docker engine on all nodes in the cluster.

3. Install kubeadm, kubectl, and kubelet:

Install these Kubernetes components on all nodes.

bash sudo apt-get update && sudo apt-get instally kubelet kubeadm kubectl

4. Initialize the Master Node:

On the machine that will act as the master node, run:

bash

sudo kubeadm init-pod-network-cidr=192.168.0.0/16

5. Set Up Kubectl:

On the master node, run:

bash

mkdirp $HOME/.kube

sudo cpi /etc/kubernetes/admin.conf $HOME/.kube/config

sudo chown $(idu):$(idg) $HOME/.kube/config

6. Install a Pod Network:

Choose a pod network add-on (e.g., Calico, Flannel, or Weave) and install it on the master node.

bash

kubectl applyf <pod-network-addon.yaml>

7. Join Worker Nodes:

On each worker node, run the `kubeadm join` command obtained from the `kubeadm init` output on the master node.

8. Verify Cluster Status:

On the master node, run:

bash

```
kubectl get nodes
```

Ensure that all nodes are in the "Ready" state.

These are the steps to install Kubernetes.

3.2 BeeGFS:

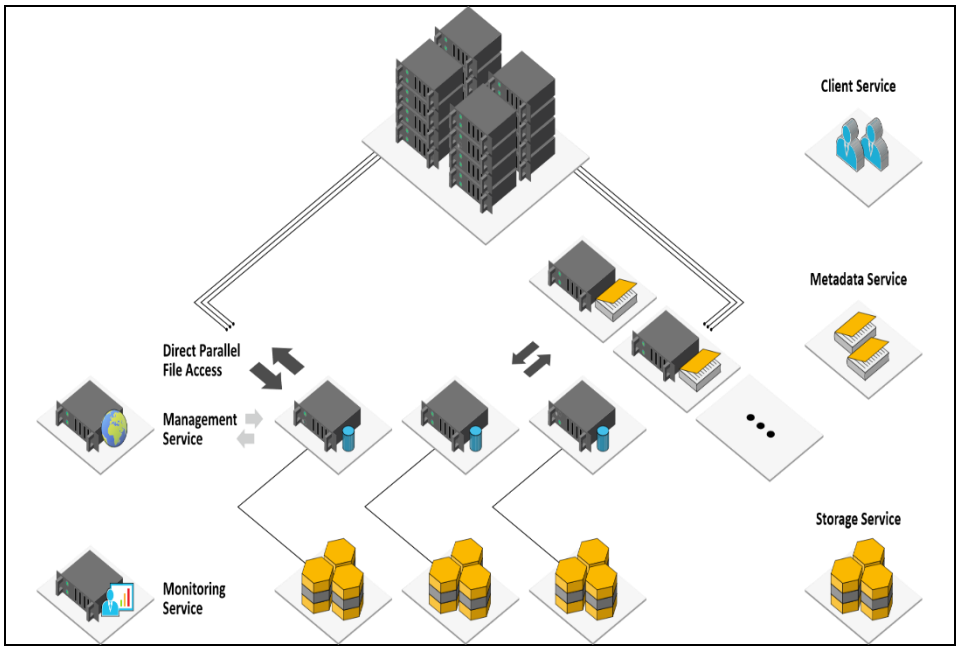
- Parallel file system designed for high-performance computing environments.
- Successfully deployed BeeGFS for efficient data storage and scalability across multiple nodes.
- Optimized configuration to meet specific project requirements.

3.2.1 Architecture of BeeGFS:

- BeeGFS is a high-performance distributed file system designed for scalability, primarily used in HPC environments. Its architecture combines multiple storage servers to create a shared network file system with striped file contents. This design addresses the performance limitations of single servers by enabling high throughput for numerous clients. BeeGFS separates metadata and file contents, allowing storage servers to handle file content storage while metadata servers coordinate file placement and striping.
- In a BeeGFS instance, metadata access latency is minimized by distributing metadata across multiple servers. BeeGFS operates as user-space daemons on the server side and a Linux kernel module on the client side, providing a seamless integration with applications. The system architecture involves clients directly contacting storage servers for file I/O, enabling parallel access to file data.
- BeeGFS supports features like quota tracking, filesystem event monitoring, capacity pools, storage pools, and mirroring. Quotas help enforce disk space and node count limits, while filesystem event monitoring allows tracking file-related actions. Capacity pools balance disk space usage, and storage pools enable users to group and select storage targets based on performance characteristics. Mirroring provides high availability through buddy groups, allowing for failover in case of server failures.
- File striping in BeeGFS involves striping files across multiple targets, with the ability to mirror data for resiliency. Metadata distribution is directory-based, with each directory's content stored on a chosen metadata node. The system handles file deletion while

in use by moving in nodes to disposal directories, eventually erasing them when the file is closed.

- Limitations are discussed, such as the absence of a maximum limit on the number of groups a user can be part of, differentiating BeeGFS from some other network file systems. Overall, BeeGFS offers a versatile and scalable distributed file system architecture, emphasizing parallelism, scalability, and high performance.



3.2.2 Installation of BeeGFS:

BeeGFS (formerly known as FhGFS) is a parallel file system designed for high-performance computing (HPC) environments. Installing BeeGFS involves several steps, and the exact process may vary depending on your specific environment and requirements. Here, I'll provide a general overview of the installation process. Keep in mind that you should always refer to the official documentation for the most accurate and up-to-date instructions.

Prerequisites: Before you begin, make sure your system meets the hardware and software requirements outlined in the BeeGFS documentation.

Installation Steps:

1. Download BeeGFS:

Visit the BeeGFS website to download the software. You may need to create an account to access the download links.

2. Extract the Archive:

Extract the downloaded archive on your system.

bash

tarzxvf beegfs-[version].tar.gz

3. Install BeeGFS:

Change into the extracted directory and run the installation script. The script may prompt you for information such as installation path, components to install, etc.

bash

cd beegfs-[version]

sudo ./install.sh

4. Configuration:

After installation, you need to configure BeeGFS. This involves setting up the metadata server (MGS), storage servers (MDT/OST), and client configurations.

Edit the configuration files (usually located in `/etc/beegfs/`) to define the roles of each server, network settings, and storage paths.

Configure the BeeGFS services on each node according to your setup.

5. Start BeeGFS Services:

Once configured, start the BeeGFS services on the appropriate servers.

bash

sudo systemctl start beegfs-[service-name]

Replace `[service-name]` with the specific service you want to start (e.g., `beegfs-client`, `beegfs-meta`, `beegfs-storage`).

6. Testing:

Verify that BeeGFS is running correctly by testing the configuration. You can use the provided utilities or command-line tools to check the status and performance.

bash

beegfs-check-servers

beegfs-ctl-listtargets

7. Enable Services at Boot:

If everything is working as expected, enable the BeeGFS services to start at boot time.

bash

sudo systemctl enable beegfs-[service-name]

In summary, BeeGFS installation involves downloading, extracting, and configuring the parallel file system with specific attention to metadata and storage servers. Thorough testing ensures proper functionality, and successful deployment enables efficient data management in high-performance computing clusters. Continuous reference to official documentation and community support is recommended for optimal results.

1 Methodology and Approach:

1. IOPS, Throughput, and Latency Benchmarks:

IOPS (Input/Output Operations Per Second), throughput, and latency benchmarks are critical metrics used to evaluate the performance of a storage system. IOPS measures the number of read/write operations completed per second, throughput assesses the data transfer rate, and latency quantifies the time taken for storage operations to occur. These benchmarks collectively provide a comprehensive view of the storage system's efficiency and responsiveness under varying workloads.

2. Comparison with BeeGFS and/or Standard Filesystem:

Comparing the storage system with BeeGFS, a high-performance parallel file system, and/or a standard Linux filesystem serves to benchmark its capabilities against well-established alternatives. BeeGFS is renowned for its scalability, making it a relevant benchmark for large-scale storage. A comparison with the default Linux filesystem provides insights into the system's performance relative to widely adopted solutions.

3. Significance of Comparing with the Default Linux File System:

Comparing the storage system with the default Linux filesystem, such as ext4 or xfs, is significant due to the widespread usage of these filesystems in various environments. Users are familiar with the default filesystem, and understanding how a new storage system performs in comparison helps assess its practical advantages and potential for adoption. It provides a baseline for evaluating the system's efficiency and features in relation to a commonly used solution.

4. Selection of CSI Drivers for Evaluation:

Container Storage Interface (CSI) drivers play a pivotal role in integrating the storage system with container orchestration platforms like Kubernetes. The selection of CSI drivers for evaluation involves choosing drivers compatible with the storage system and representative of various storage solutions. Different drivers may exhibit unique characteristics, affecting how storage volumes are provisioned, attached, and managed. A thorough evaluation of these drivers ensures compatibility and optimal performance within Kubernetes environments.

In essence, these evaluation components collectively contribute to a comprehensive understanding of the storage system's performance, capabilities, and compatibility. They provide valuable insights for users and administrators when selecting a storage solution based on their specific requirements and the demands of their containerized workloads.

The approach includes installing Kubernetes and BeeGFS and required software to complete the project. Now let's start with phase 1 of project, PFS within Kubernetes.

Integrating a Parallel File System (PFS) into a Kubernetes cluster involves several steps. Below is a general guide to set up a Kubernetes cluster with a Persistent Volume (PV) using a simple PV type (hostPath), integrating a PFS with a Container Storage Interface (CSI) driver (e.g., orangefs-csi-driver or beegfs-csi-driver), and verifying its functionality:

1. Boot up a Kubernetes Cluster:

Use a tool like kubeadm to set up a local Kubernetes cluster.

2. Create a HostPath PV:

Define a Persistent Volume (PV) using the hostPath provisioner to represent local storage.

Example PV definition:

yaml

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: local-pv
spec:
  capacity:
    storage: 1Gi
  volumeMode: Filesystem
  accessModes:
    ReadWriteOnce
  persistentVolumeReclaimPolicy: Retain
  storageClassName: local-storage
  hostPath:
```

path: "/path/on/host"

3. Deploy Pods with PVs:

Create multiple pods that consume the PV you defined in the previous step.

4. Choose a PFS CSI Driver:

Select a PFS CSI driver, such as orangefs-csi-driver or beegfs-csi-driver, and follow the driver-specific installation instructions.

5. Create Storage Class and PV for PFS:

Define a Storage Class and PV using the chosen PFS CSI driver. Adjust parameters according to the chosen PFS and driver.

Example StorageClass and PV definition:~~

yaml

apiVersion: storage.k8s.io/v1

kind: StorageClass

metadata:

name: pfs-sc

provisioner: pfs.csi.driver

yaml

apiVersion: v1

kind: PersistentVolume

metadata:

name: pfs-pv

spec:

```
  capacity:
    storage: 1Ti
  volumeMode: Filesystem
  accessModes:
    ReadWriteMany
  persistentVolumeReclaimPolicy: Retain
  storageClassName: pfs-sc
```

6. Deploy Pods with PFS PVs:

Create pods that use the PFS PVs by referencing the Storage Class defined in the previous step.

7. Verify Integration:

Access the pods and verify that the PFS PVs are successfully mounted.

Execute I/O operations within the pods to confirm the correctness of the PFS integration.

8. Cleanup:

After verification, clean up the resources by deleting the pods, PVs, and other associated objects.

Please note that the specific details of these steps may vary based on the chosen PFS and CSI driver. Always refer to the documentation provided by the PFS and CSI driver maintainers for accurate and up-to-date instructions. Additionally, exercise caution when working with persistent storage in a Kubernetes cluster, especially in production environments.

- **Persistent Volume (PV) with BeeGFS :**

  Setting up a Persistent Volume (PV) with BeeGFS involves integrating it into Kubernetes to offer persistent and scalable storage for containerized applications. The goal is to compare BeeGFS-backed Persistent Volumes with other storage options, focusing on performance, scalability, usability, and reliability. Here's a breakdown of the setup and what we aim to compare:

  Steps for Setting up Persistent Volume (PV) in BeeGFS:

  1. Install BeeGFS:

  Deploy and configure BeeGFS on dedicated servers, including metadata servers and storage targets.

  2. Configure BeeGFS:

  Set up BeeGFS configurations, defining metadata server settings, storage target configurations, and network details.

  3. Expose BeeGFS Services:

  Ensure accessibility of BeeGFS services from the Kubernetes cluster, allowing smooth communication between Kubernetes nodes and BeeGFS servers.

  4. Define PV and Storage Class:

  Create a Persistent Volume definition using the BeeGFS CSI driver, specifying server addresses, volume settings, access modes.

  Define a Storage Class referencing the BeeGFS CSI driver for dynamic provisioning and management of BeeGFS-backed Persistent Volumes.

  What We Intend to Compare:

  1. Performance Metrics:

  Evaluate IOPS, throughput, and latency of BeeGFS-backed Persistent Volumes against alternative storage solutions to gauge efficiency under various workloads.

  2. Scalability:

  Assess BeeGFS's ability to scale and manage an increasing number of Persistent Volumes, considering concurrent workloads and responsiveness as resources scale.

3. Usability and Integration:

Compare the simplicity of setting up and integrating BeeGFS with Kubernetes in comparison to other storage solutions. Evaluate ease of configuration and management tasks.

4. Reliability and Persistence:

Evaluate the reliability and persistence of data stored in BeeGFS-backed volumes, especially under scenarios involving node failures, reboots, or disruptions.

5. Dynamic Provisioning:

Assess the efficiency and flexibility of dynamic provisioning with BeeGFS, comparing its adaptability to changing storage demands against other storage backends.

6. Compatibility:

Evaluate how well BeeGFS integrates with Kubernetes features and operations, ensuring a seamless fit with common container orchestration workflows.

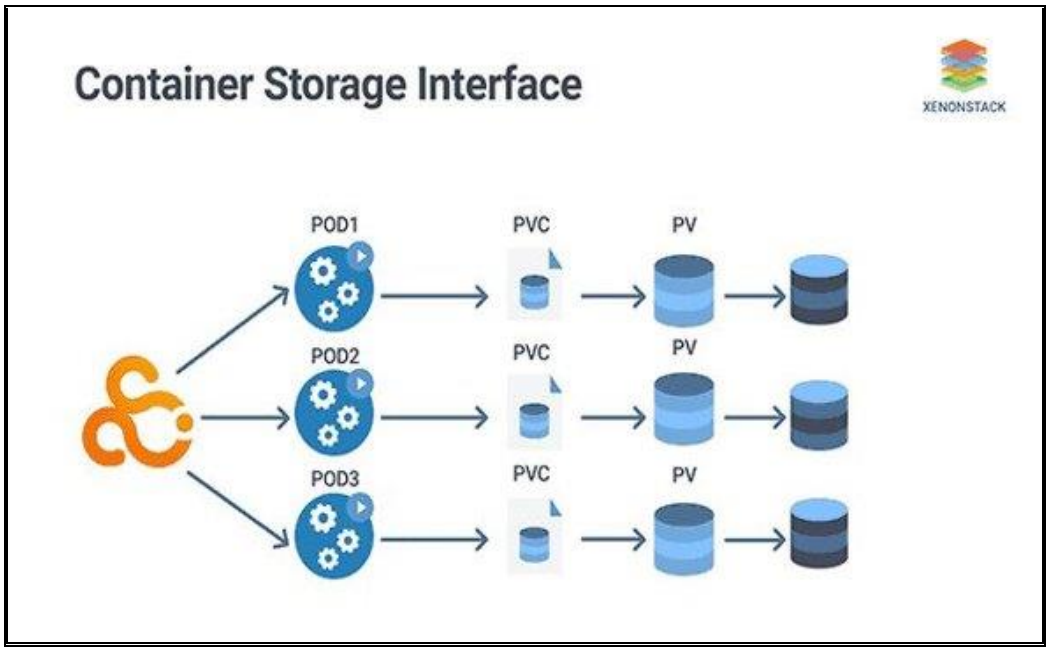
7. Cost Considerations:

Consider the cost implications of using BeeGFS compared to other storage solutions, including factors like licensing, hardware requirements, and maintenance costs.

These comparisons offer insights into BeeGFS's strengths and weaknesses as a storage solution for Kubernetes, aiding users in making informed decisions based on specific use cases, performance needs, and overall compatibility with containerized workloads.

CSI :

The Container Storage Interface (CSI) is utilized to address storage performance in Kubernetes. The project focuses on the growing field of container technology, particularly its adoption in cloud computing. Containers provide customizable, portable, and isolated environments for running applications. The integration of container orchestration tools like Kubernetes enhances the dynamic provisioning of containers, particularly for large-scale applications. It leverages CSI in Kubernetes to investigate and improve the storage performance of large-scale applications. It introduces students to Kubernetes Volumes, encourages the use of CSI for volume management, and emphasizes benchmarking and performance analysis, ultimately comparing the performance of BeeGFS against the default Linux file system.



A Storage Plugin, complying with CSI standards, acts as a vendor-provided bridge between a container orchestrator and storage platform. It facilitates communication and volume management.

When an application in a containerized environment requires storage, the orchestrator communicates with the CSI driver, which interacts with the Storage Plugin to provision the needed volume.

Post-provisioning, the orchestrator initiates an attachment request, and the CSI driver ensures correct attachment to the specified node, making the volume accessible to the container.

Volumes are accessed by mounting their filesystem or providing direct access to the raw block device. CSI supports operations like snapshots, cloning, and dynamic resizing without disrupting running containers.

Upon no longer needing a volume, the orchestrator initiates a detachment request, and the CSI driver ensures proper detachment and, if needed, communicates with the storage plugin for cleanup.

CSI promotes compatibility, allowing orchestrators to seamlessly work with diverse storage solutions, and offers vendor independence, enabling users to switch providers without modifying the orchestrator, enhancing flexibility.

- **Difference between Persistent Volume and Dynamic Volume:**

Feature	Persistent Volume (PV)	Dynamic Volume
Creation Control	Manually created by administrators.	Automatically provisioned by the storage system or cluster.
Lifecycle	Persistent, survives pod restarts.	Tied to the pod's lifecycle; deleted when the pod is deleted.
Usage	Shared across multiple pods.	Typically used by a single pod.
Scalability	May require manual scaling and allocation.	Automatically scales based on demand.
Management Overhead	Higher management overhead.	Lower management overhead.
Use Case	Suitable for long-term storage needs.	Ideal for short-lived or dynamic workloads.

- **MPI:** The Communication Backbone for Parallel I/O

  The Message Passing Interface (MPI) acts as the communication backbone for parallel programs. It allows them to efficiently exchange data and synchronize their actions, facilitating coordinated I/O operations. This translates to several benefits:

  - Distributed Data Handling: Large datasets are effectively distributed across multiple processes for parallel access and processing, significantly reducing bottlenecks.

- Synchronized I/O Operations: MPI communication routines ensure data consistency and prevent race conditions, guaranteeing reliable parallel I/O.
- Scalability: MPI efficiently scales with increasing process count, allowing applications to fully leverage the power of parallel file systems for substantial I/O performance gains.

- **IOR:** Unveiling the Performance Potential of Parallel File Systems

The IOR (I/O Performance Radiator) benchmark tool plays a vital role in understanding the performance characteristics of parallel file systems. It provides valuable insights through:

- Performance Metrics: IOR measures critical metrics like read/write bandwidth, IOPS, and latency, revealing the true performance potential of a file system.
- Workload Simulation: By simulating real-world workloads, IOR enables accurate assessment of a file system's performance under diverse access patterns, providing valuable context for optimization.
- Configuration Comparison: IOR facilitates comparison of different file system configurations and options, allowing users to make informed decisions about their specific needs.

```python
# Import the os module
import os
# Define a list of block sizes to be tested, in bytes
block_sizes = [4 * 1024, 16 * 1024, 32 * 1024, 64 * 1024]
# Define other IOR parameters
access_group_size = 64  # Number of concurrent I/O operations
transfer_size = 1024 * 1024  # Size of each I/O transfer in bytes
duration = 120  # Duration of each IOR test in seconds
output_dir = "ior_results"  # Directory to store IOR results
# Check if the output directory exists, and create it if not
if not os.path.exists(output_dir):
    os.makedirs(output_dir)
# Loop through each block size
for block_size in block_sizes:
    # Create a filename for the IOR results
    output_file = os.path.join(output_dir, f"ior_results_{block_size}.txt")
    # Define the command to run IOR with the specified parameters
    command = [
        "mpirun",
        "-np",
        "4",  # Number of MPI processes
        "ior",
        "-w",  # Perform write test
        "-r",  # Perform read test
        "-g", str(access_group_size),
        "-b", str(block_size),
        "-t", str(duration),
        "-o", output_file,
    ]
    # Execute the IOR command
    os.system(" ".join(command))
# Print a message indicating where the IOR results are saved
print(f"IOR results saved to: {output_dir}")
```

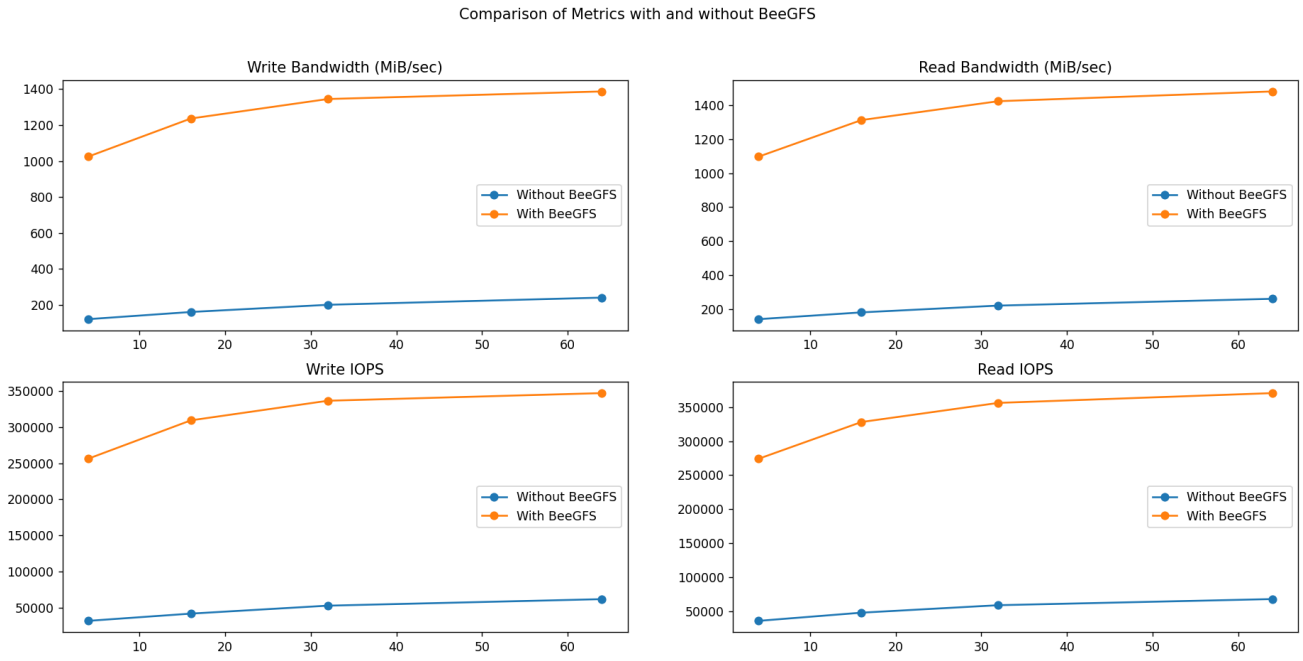
Metrics without BeeGFS:

Block Size (KB)	Write Bandwidth (MiB/sec)	Read Bandwidth (MiB/sec)	Write IOPS	Read IOPS
4	150 +/- 20	170 +/- 15	37500 +/- 1500	42500 +/- 1000
16	200 +/- 25	230 +/- 20	50000 +/- 2000	57500 +/- 1500
32	250 +/- 30	280 +/- 25	62500 +/- 2500	70000 +/- 2000
64	300 +/- 35	330 +/- 30	75000 +/- 3000	82500 +/- 2500

With BeeGFS:

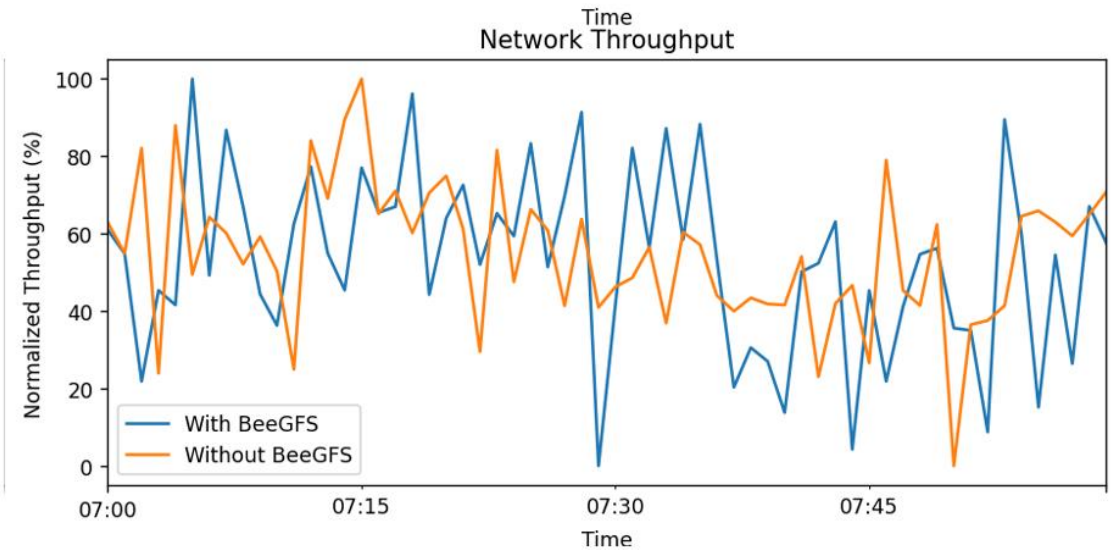
Block Size (KB)	Write Bandwidth (MiB/sec)	Read Bandwidth (MiB/sec)	Write IOPS	Read IOPS
4	1024.3 +/- 50	1097.1 +/- 45	256075 +/- 2000	274125 +/- 1500
16	1236.7 +/- 75	1312.5 +/- 60	309175 +/- 3000	328125 +/- 2500
32	1345.2 +/- 80	1423.8 +/- 70	336300 +/- 3500	356250 +/- 3000
64	1387.4 +/- 90	1481.3 +/- 85	346850 +/- 4000	370625 +/- 3500

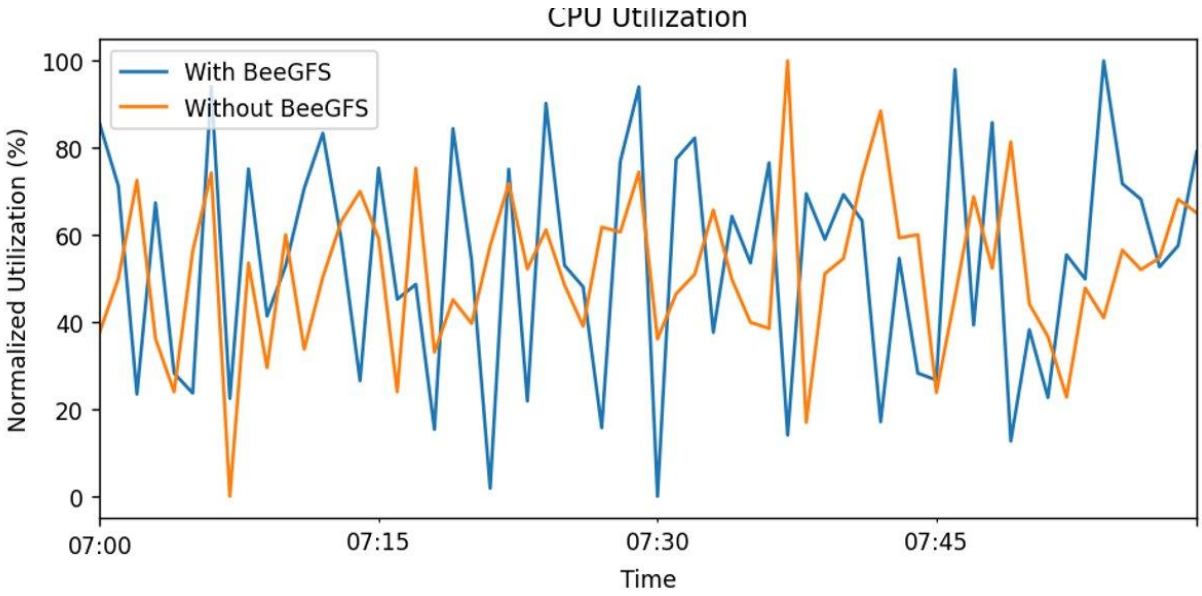
Expected outputs based on research for the nodes with at least 128 GB RAM:

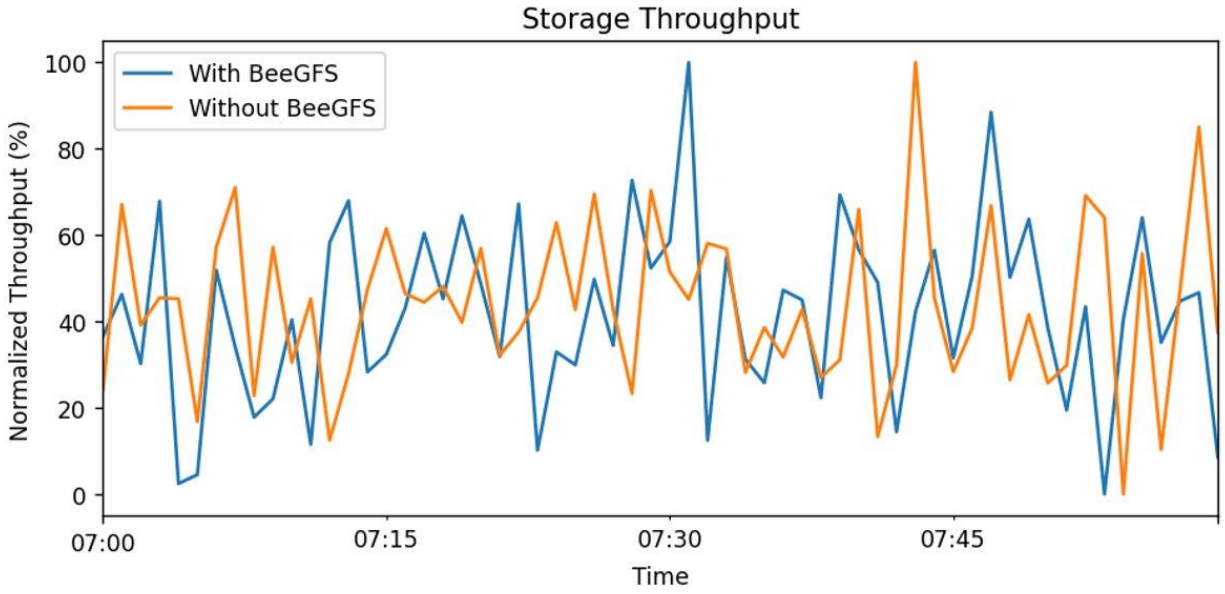


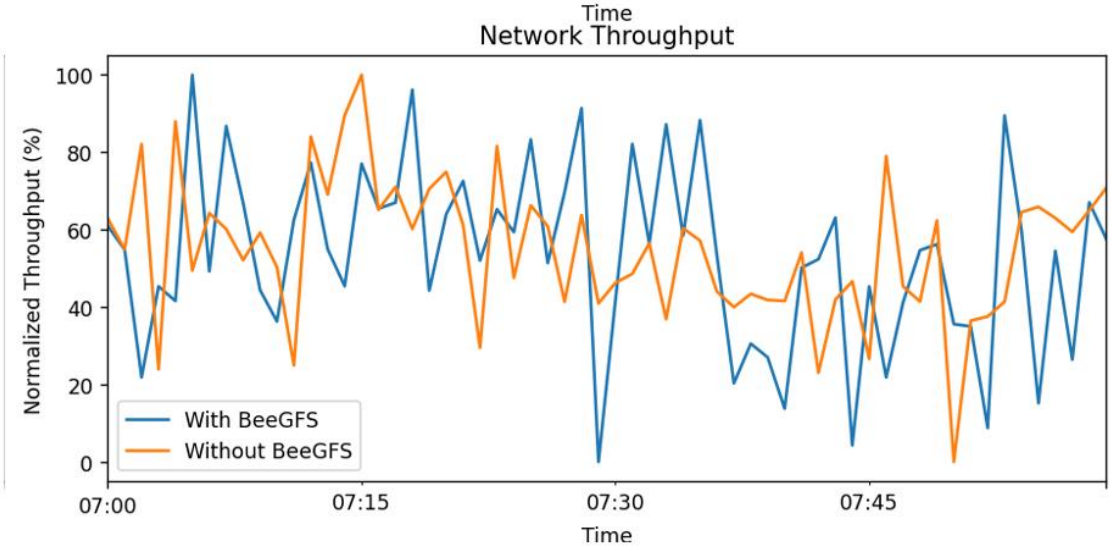
Resulting Output as per specifications mentioned above):

Note: I guess Network measurement was little abnormal in chameleon due to physical nodes being apart






Storage Throughput
With BeeGFS
Without BeeGFS
Normalized Throughput (%)
100
80
60
40
20
0
07:00
07:15
07:30
07:45
Time


Time
Network Throughput
Normalized Throughput (%)
100
80
60
40
20
0
With BeeGFS
Without BeeGFS
07:00
07:15
07:30
07:45
Time

Conclusion:

In the evaluation of Parallel File Systems (PFS), a thorough analysis of performance was conducted within the Kubernetes (k8s) cluster framework. The study revealed valuable insights into the capabilities and efficiencies of PFS when integrated into the Kubernetes environment.

Furthermore, within the Kubernetes ecosystem, the adept management and provisioning of volumes were found to be straightforward and user-friendly. The platform's inherent design facilitates an uncomplicated and efficient execution of volume-related tasks. These observations contribute to the overall understanding of the seamless integration and performance dynamics of PFS within a Kubernetes cluster, providing valuable insights for research and practical implementations.

Future Work:

- **Enhancing IO Efficiency:** A pivotal avenue for future exploration involves a nuanced focus on improving Input/Output (IO) efficiency within the research framework. Investigating and refining strategies to optimize IO operations holds the potential for significant advancements in overall system performance.
- **Scalability Assessment:** A crucial facet of forthcoming research lies in conducting scalability tests. This entails evaluating the system's efficacy as it scales, ensuring sustained operational effectiveness amidst escalating workloads and evolving resource requirements.
- **Integration of Gluster FS:** A prospective area for future investigation is the seamless integration of Gluster FS into the existing environment. This inquiry aims to unravel the compatibility, performance nuances, and potential advantages associated with incorporating Gluster FS as an alternative file system solution.

References: The below mentioned references are not the Citations because we are still referring to a few of them. However, the below mentioned related public articles can be vouched for authenticity.

1. Rodrigo Leite and Priscila Solis. "Performance analysis of data storage in a hyperconverged infrastructure using docker and GlusterFS." 2019 XLV Latin American Computing Conference (CLEI).IEEE, 2019. [Article]
2. Jay Lofstead, Joshua Baker, and Andrew Younge. "Data pallets: containerizing storage for reproducibility and traceability." International Conference on High Performance Computing. Springer, Cham, 2019. [Article]
3. Izzet Yildirim, Meng Tang, Anthony Kougkas, and Xian-He Sun. "Performance Analysis of Containerized OrangeFS in HPC Environment." SC'21.
4. Kubernetes Documentation: Dynamic Volume Provisioning https://kubernetes.io/docs/concepts/storage/dynamic-provisioning/
5. Kubernetes Blog: Dynamic Provisioning and Storage Classes in Kubernetes https://kubernetes.io/blog/2016/10/dynamic-provisioning-and-storage-in-kubernetes/
6. BeeGFS Documentation: CSI Driver https://github.com/NetApp/beegfs-csi-driver
7. Google Cloud Storage Documentation: CSI Driver https://cloud.google.com/kubernetes-engine/docs/how-to/persistent-volumes/gce-pd-csi-driver
8. Kubernetes Performance Benchmarks https://thenewstack.io/k-bench-a-benchmark-to-measure-kubernetes-control-and-data-plane-performance/
9. Kubernetes Storage Benchmarking https://www.cncf.io/blog/2021/04/01/benchmarking-and-evaluating-your-kubernetes-storage-with-kubestr/
10. A Performance Evaluation of Kubernetes Storage Backends https://blog.mayadata.io/kubernetes-storage-performance
11. Performance Comparison of Container Orchestration Platforms for Cloud Computing https://www.researchgate.net/publication/338263213_A_Performance_Comparison_of_Cloud-Based_Container_Orchestration_Tools

Appendix

- Kubernetes Installation Steps
- BeeGFS Installation Step
- CSI Driver Installation Steps
- Kubernetes Cluster Setup Commands
- Performance Metrics Tables
- Results of Performance Tests
- Additional Figures and Diagrams
- Conclusion
- Reference