GEORG-AUGUST-UNIVERSITÄT GÖTTINGEN IN PUBLICA COMMODA SEIT 1737

HPS

Seminar Report

Containers in HPC

Utkarsh Pathak

MatrNr: 11175062

Supervisor: Anila Ghazanfar

Georg-August-Universität Göttingen
Institute of Computer Science

March 31, 2026

Abstract

Container technologies have made their way into High-Performance Computing (HPC) environments in a big way. The reason is pretty clear: people want reproducibility, portability, and isolation for their workloads. However, the question of whether containers introduce measurable performance overhead, particularly for tightly coupled parallel applications. This study addresses that question through systematic empirical evaluation.

We ran a systematic performance comparison of three container runtimes: Apptainer 1.4.5 (the former Singularity), Podman 5.8.0, and Charliecloud 0.38 on a three-node Simple Linux Utility for Resource Management (SLURM) cluster built from virtualised AMD EPYC nodes. We looked at performance from three angles:

- **Message Passing Interface (MPI) communication:** By using OSU Micro-Benchmarks v7.4. [Net]
- **Input/Output (I/O) throughput:** By using Interleaved Or Random (IOR) Benchmark.[Law]
- **Filesystem and Metadata Performance:** Was tested using MDTest.[Law]

For each runtime, we compared against a native bare-metal Slurm baseline to figure out exactly what overhead, if any, the container layer introduces. We also kept an eye on CPU and memory usage throughout all the benchmark runs to get a full picture of resource efficiency.

The key findings are as follows. All three runtimes delivered near-native MPI communication performance, point-to-point latency was generally within 10% across most message sizes. In parallel I/O, write bandwidth was pretty much the same across the board, hovering around 23.5–23.8 MiB/s. Read performance, though, was a completely different story: Podman gave us twice the read throughput of bare metal, while Apptainer and Charliecloud took four times longer to read the same data. That seems to come down to how each runtime's mount namespace interacts with the underlying storage stack. On the metadata side, Charliecloud handled file stat operations 16.2% faster than bare metal. When we looked at CPU usage, all runtimes were identical during MPI workloads (a steady 25% per node). But during I/O phases, Podman's CPU usage stayed much higher for a long stretch, thanks to its FUSE-based overlay filesystem. Memory-wise, Apptainer had the biggest footprint, about 120 MB above bare metal, which we traced back to its SquashFS loopback mount buffer.

These results demonstrate that runtime selection has a measurable impact on performance, and that the optimal choice depends on the dominant workload characteristics. This report provides empirical evidence to guide that decision.

Declaration on the use of ChatGPT and comparable tools in the context of examinations

In this work I have used ChatGPT or another AI as follows:

- ☐ Not at all
- ☐ During brainstorming
- ☐ When creating the outline
- ☑ To write individual passages, altogether to the extent of 10% of the entire text
- ☐ For the development of software source texts
- ☐ For optimizing or restructuring software source texts
- ☑ For proofreading or optimizing
- ☐ Further, namely: -

I hereby declare that I have stated all uses completely.
Missing or incorrect information will be considered as an attempt to cheat.

Contents

List of Tables v

List of Figures v

List of Listings v

List of Abbreviations vi

1 Introduction 1

2 Background 1
- 2.1 High-Performance Computing and Job Scheduling 1
- 2.2 Containerization Technologies 2
- 2.3 HPC-Oriented Container Runtimes 2
  - 2.3.1 Podman 2
  - 2.3.2 Apptainer 3
  - 2.3.3 Charliecloud 3

3 Experimental Setup 4
- 3.1 Cluster Architecture 4
- 3.2 Hardware and Software Specifications 4
  - 3.2.1 Container Images 5
- 3.3 Benchmark Suite 5
- 3.4 Experimental Methodology 6

4 Results and Analysis 7
- 4.1 OSU Micro-Benchmarks 7
  - 4.1.1 Point-to-Point Latency (osu_latency): 7
  - 4.1.2 Point-to-Point Bandwidth (osu_bw): 8
  - 4.1.3 Bi-directional Bandwidth (osu_bibw): 9
  - 4.1.4 Allreduce (osu_allreduce): 9
  - 4.1.5 OSU Wall Clock Time (sec) 9
- 4.2 IOR Parallel I/O Benchmarks 10
- 4.3 MDTest Metadata Benchmarks 10
- 4.4 CPU and Memory Utilization 11
  - 4.4.1 OSU Benchmark CPU and Memory Utilization 11
  - 4.4.2 IOR and MDTest Benchmark CPU and Memory Utilization 11

5 Discussion 19
- 5.1 MPI communication 19
- 5.2 I/O and metadata 19
- 5.3 CPU and memory 20
- 5.4 Runtime Selection Guidelines 20
- 5.5 Limitation 20

6 Conclusion **20**
6.1 Key Findings . . . 20
6.2 Practical Implications for HPC Practitioners . . . 21

References **22**

A Code samples **A1**

List of Tables

1 Per-node hardware specifications. 5
2 Software Versions. 6
3 OSU MPI Point-to-Point Latency (ms). 7
4 OSU MPI Point-to-Point Bandwidth (MB/s). 9
5 OSU MPI Bi-Directional Bandwidth (MB/s). 10
6 OSU Allreduce Latency (ms). 12
7 OSU Benchmark Wall clock Time (seconds) 12
8 IOR Write Test Results. 13
9 IOR Read Test Results. 14
10 MDTest Mean Metadata Operation Rates (ops/sec). 14
11 MDTest Detailed Statistics (ops/sec). 15

List of Figures

1 Podman Architecture 3
2 Apptainer Architecture 4
3 CharlieCloud Architecture 5
4 Cluster topology diagram 6
5 OSU osu_latency 8
6 OSU osu_bw 8
7 OSU osu_bibw 11
8 OSU osu_allreduce 13
9 IOR write Results 13
10 IOR Read Results 14
11 OSU SLURM CPU Utilization 15
12 OSU Podman CPU Utilization 16
13 OSU Apptainer CPU Utilization 16
14 OSU CharlieCloud CPU Utilization 16
15 OSU all runtime Memory Utilization 17
16 IOR and MDTest SLURM CPU Utilization 17
17 IOR and MDTest Podman CPU Utilization 17
18 IOR and MDTest Apptainer CPU Utilization 18
19 IOR and MDTest CharlieCloud CPU Utilization 18
20 IOR and MDTest all Memory Utilization 19

List of Listings

List of Abbreviations

HPC High-Performance Computing

I/O Input/Output

OS Operating System

MPI Message Passing Interface

SLURM Simple Linux Utility for Resource Management

conmon Container Monitor

OverlayFS Overlay Filesystem

PMI/PMIx Process Management Interface

IOR Interleaved Or Random

POSIX Portable Operating System Interface

OCI Open Container Initiative

SLIRP Serial Line Internet Protocol

1 Introduction

HPC systems are created to execute computationally intensive or data intensive workloads over parallel hardware which comprise of multiple processors, high speed interconnects and distributed storage systems. HPC applications traditionally rely on low-level libraries to extract maximum performance. From a performance perspective this method is highly effective but as the software complexity increases, it also brings significant challenges such as portability, reproducibility and dependency management.

Containerization Technologies were originally developed for cloud-native workloads. They encapsulate an application along with its entire runtime environment within a portable image. This helps to get consistent results in a heterogeneous system. This fundamental concept of Containers can solve the problem in traditional HPC applications.

Tools such as Docker are widely in the cloud industry, but their daemon-based architecture, and requirement of root-level privileges make it a bad choice for the traditional multi-tenant, security conscious HPC environment. To address these problems, various community driven projects such as Podman, Apptainer, Charliecloud emerged, each with different ideologies to be suitable for HPC.

This study investigates the following:

1. Does containerization introduce measurable performance overhead when using host MPI communication?

2. How does containerization perform in I/O workloads?

3. Is there a difference in CPU and memory resource utilization between containerized and native execution?

To evaluate this a three node SLURM managed cluster was created, each node consisting of 4 virtualized vCPUs, 8 GB RAM, connected to each other via Ethernet and running Ubuntu 22.04 LTS as Operating System (OS).

This report is structured into sections covering background, experimental methodology, benchmark results, discussion, and conclusion.

‘

2 Background

2.1 High-Performance Computing and Job Scheduling

HPC refers to the use of aggregated computational resources, typically clusters of interconnected compute nodes to solve problems that exceed the capacity of a single machine. HPC workloads are characterized by high degrees of parallelism, intensive data exchange between processes, and demanding I/O patterns.

The MPI is the dominant programming model for distributed-memory parallelism in HPC. MPI processes communicate by explicitly passing messages across a network interconnect [MPI23], this makes the latency and bandwidth of the network a critical part of application performance.

Job scheduling in HPC clusters is typically managed by a workload manager. The SLURM is one of the most widely deployed schedulers in production HPC environments[YJG03]. SLURM allocates compute nodes, manages job queues, enforces resource policies, and provides a standardized job submission interface. In this study, SLURM version 21.08.5 [Sch] is used to orchestrate all benchmark runs, ensuring consistent resource allocation across experiments.

2.2 Containerization Technologies

Containerization is a form of virtualization in which applications and their dependencies are packaged into isolated units called containers. Unlike traditional hypervisor-based virtual machines, containers share the host operating system kernel, this results in lower kernel overhead. Containers give us process isolation, file-system namespace separation, and controlled resource access, this makes it suitable for reproducible and portable software deployment.

The Open Container Initiative (OCI) specification which is maintained by the Linux Foundation [Ope], defines standard formats for container images and runtimes, this enables interoperability between different container tools. Docker is largely used in cloud and web contexts, its requirement for a root-owned daemon process makes it incompatible with the security model of shared HPC systems, where users operate without elevated privileges.[Doc]

2.3 HPC-Oriented Container Runtimes

The three container runtimes which are evaluated in this study are designed specifically to address the weakness of runtimes such as Docker when deployed in HPC environments. Even though they share a common goal of being unprivileged, daemonless containerization, they differ in their internal architecture, image formats, security models, filesystem handling, and MPI integration strategies. This section provides a detailed account of each runtime's design and operation, followed by a comparative summary.

2.3.1 Podman

Podman is a daemonless container runtime technology developed by Red Hat initially but now is maintained by the containers community (containers.io). It is completed compliant with the OCI specification for both image format and runtime. [Pod]

Each podman run or podman exec call is a self-contained , it sets up namespaces, resolves the image, and launches the container without any coordinating service [Pod].

Podman by default stores images in a layered format using an Overlay Filesystem (OverlayFS) drivers, typically in the user home directory. [The].

In rootless mode, Podman uses Linux user namespaces to create an isolated execution environment without any elevated privileges. The user namespaces remaps the container's internal root UID to the invoking user's UID on the host, this allows the container to perform operations that appear privileged (inside the container runtime) while remaining unprivileged from the host's perspective. Podman uses slirp4netns. slirp4netns a user-space network stack implementation based on Serial Line Internet Protocol (SLIRP) that provides NAT-based networking for containers without requiring kernel-level network namespace privileges. [Sud]

Podman uses Container Monitor (conmon) as a per-container monitoring process. conmon exists only for the lifetime of its associated container and runs unprivileged.[Con]

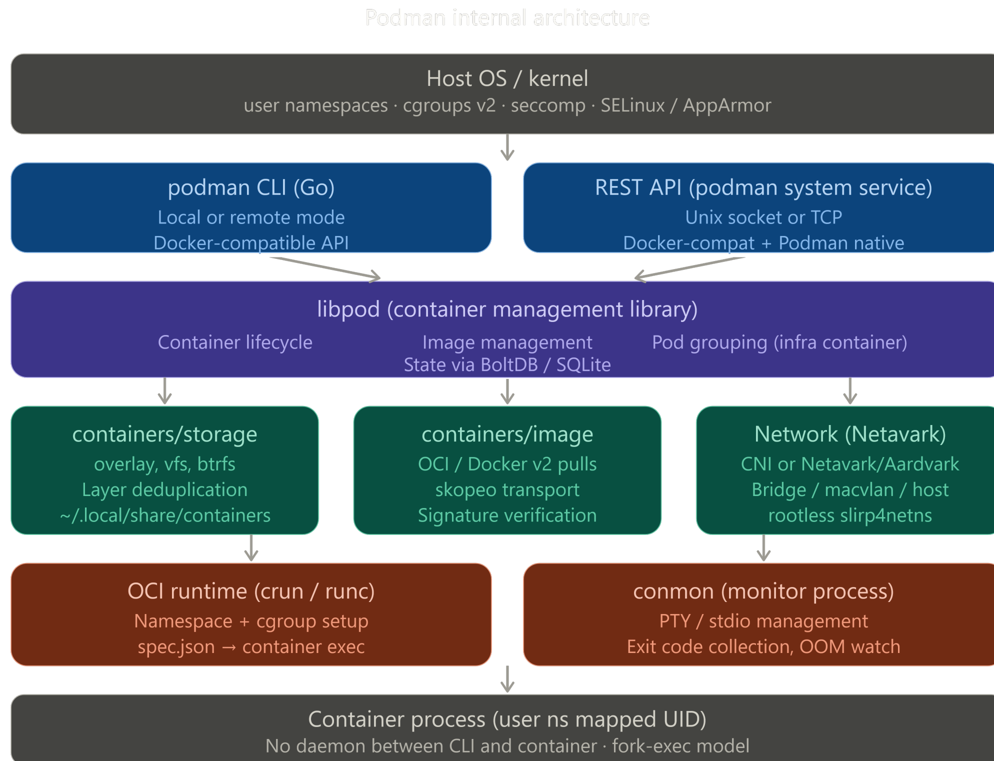

Figure 1: Podman Architecture, based on [Pod]

For MPI workloads, the recommended approach is to configure Podman to use host network mode (–network=host), this bypasses slirp4netns entirely and gives the container direct access to the host network interfaces and stack.

2.3.2 Apptainer

Apptainer, formerly Singularity. It was developed by Gregory Kurtzer at Lawrence Berkeley National Laboratory in 2015 [KSB17], it is now maintained by the Linux Foundation. Version 1.4.5, used in this study [App].

Apptainer uses a single read-only file which holds the entire file system and metadata.It automatically bind-mounts several host paths at startup such as the user home directory, host machines **/tmp** and **/var/tmp** to the container [App].

Apptainer is daemon-less and it either relies on Linux user namespaces or setuid.

Apptainer supports both bind-mount the host MPI libraries and a hybrid approach in which the hosts **mpirun!** (**mpirun!**) is injected in the container enviornment.

2.3.3 Charliecloud

Background and History

Charliecloud was developed by Reid Priedhorsky and Tim Randles at Los Alamos National Laboratory (LANL) and first published in 2017 [PR17]. Charliecloud's image

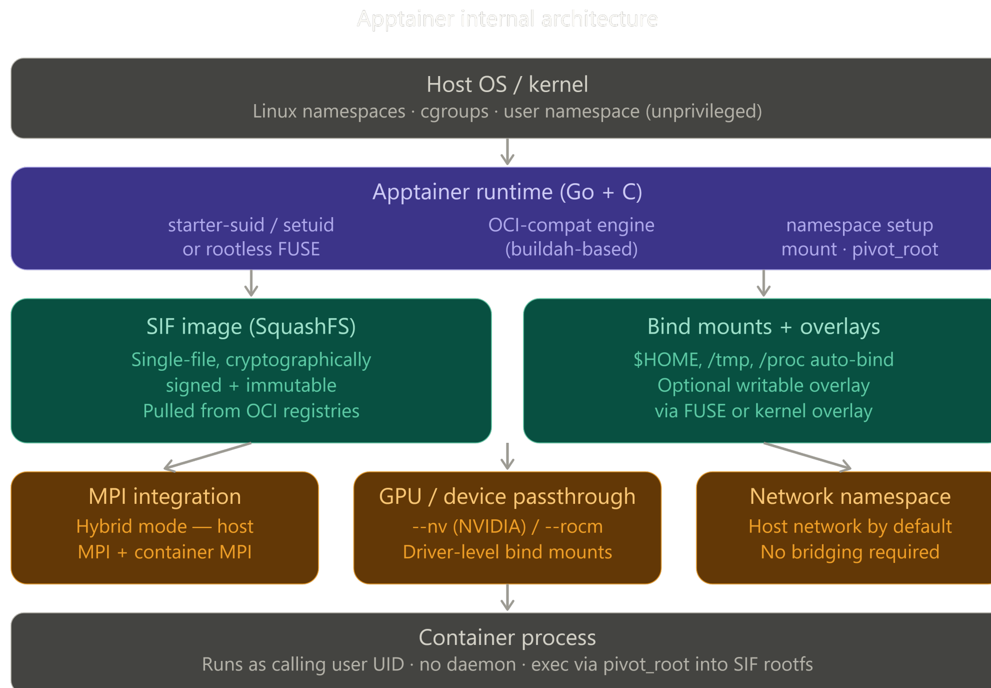

Figure 2: Apptainer Architecture, based on [App]

is a standard, unpacked Portable Operating System Interface (POSIX) directory tree. There are no layers, no manifest files, no compression [PR17].

The **ch-run** create user namespace, create a mount namespace, mounting the container flat directory tree to a new root and performing a **pivot_root** or **chroot** operation to change the filesystem root [PR17]. The host MPI is used is shared with the container runtime.

3 Experimental Setup

3.1 Cluster Architecture

A three-node SLURM cluster was configured with one controller and two worker nodes. All nodes are connected to each other with Ethernet network and a common NFS storage managed by controller is used as storage. The topology is illustrated in fig. 4.

3.2 Hardware and Software Specifications

All three nodes are virtualized instances with similar hardware, this is done to ensure uniform execution environment across the cluster. The hardware specification for each node is as follows in table 1. Software versions are listed in table 2, all componenents remained constant across experiments except the container runtimes.

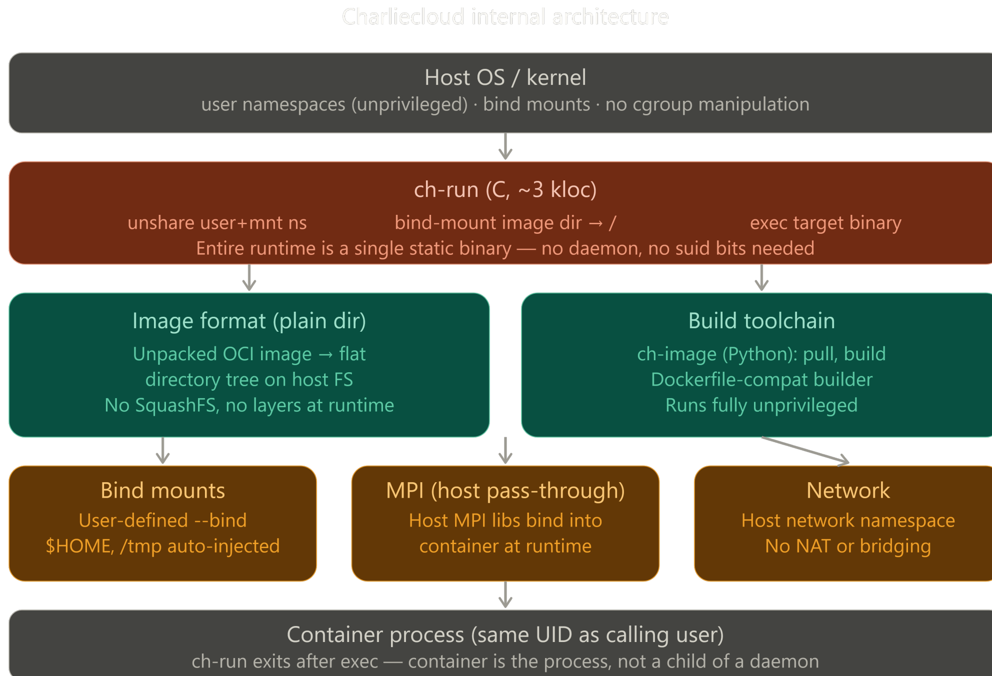

Figure 3: CharlieCloud Architecture, based on [PR17]

Table 1: Per-node hardware specifications.

Component	**Specification**
CPU Model	AMD EPYC, CPU Family 23
CPU Cores	4 vCPUs
RAM	8GB
Network	Ethernet

3.2.1 Container Images

All container images are created from Ubuntu 22.04 base image[Can22], along with Open MPI 4.1.2 [Gab+04]. There are two versions of the images one including OSU-Mirco Benchmarks, and the other including IOR-MDTest compiled and installed in the images.

For Apptainer, images were built in the **.sif** format. Podman used OCI-compliant images. Finally Charliecloud images were flattened into the unpacked directory tree by using the **ch-convert** command on the Podman images.

For all three container runtimes, external MPI (hybrid MPI) was used. The host machines Process Management Interface (PMI/PMIx) interface is injected inside the container images.

3.3 Benchmark Suite

Three benchmark tools were used to evaluate performance, with all tests configured to measure inter-node communication.

- **OSU Mirco-Benchmarks**: point-to-point latency (osu_latency), bandwidth (osu_bw), bidirectional bandwidth (osu_bibw), and collective latency (osu_allreduce).[Net]

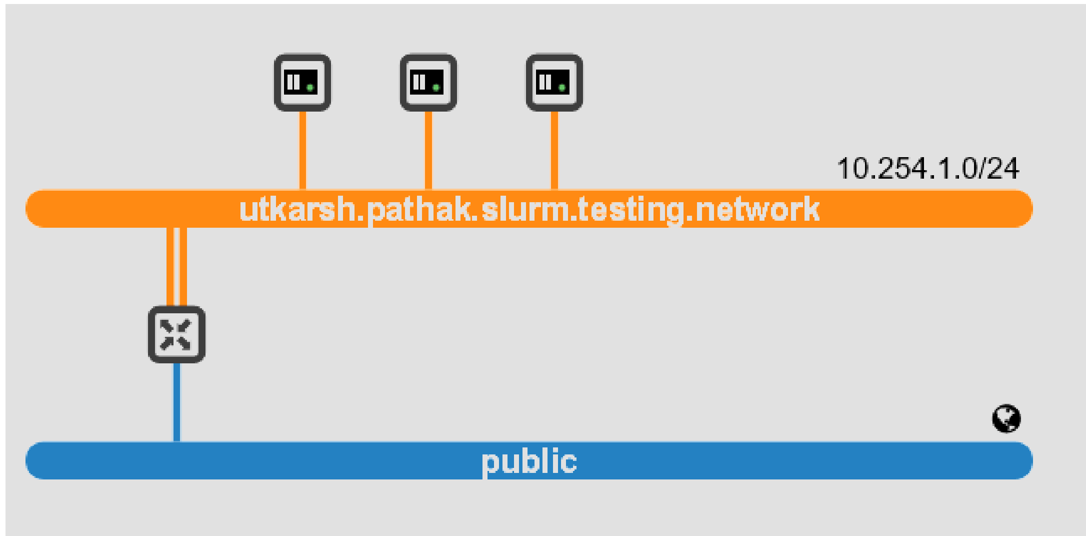

Figure 4: 1 controller node, 2 worker nodes, connected via Ethernet switch

Table 2: Software Versions.

Component	**Version**
Operating System	Ubuntu 22.04 LTS
Linux Kernal	Linux 5.15.0-171-generic
Workload Manager	Slurm (slurm-wlm) 21.08.5
MPI Implementation	OpenMPI 4.1.2
Apptainer	1.4.5
Podman	5.8.0
Charliecloud	0.38
OSU Micro-Benchmarks	v7.4
IOR / mdtest	4.1.0+dev

- **IOR**: Parallel I/O throughput. A write phase followed by a read phase, each with 1 MiB transfer size, 16 MiB block size, and 16 segments per process (256 MiB total I/O per process). The file was retained after writes to avoid page-cache effects during reads. [Law]
- **MDTest**: Metadata performance. Each process created, stated, and removed 1000 files, with three iterations. Unique working directories per process avoided contention, and leaf-level operations were used.

3.4 Experimental Methodology

SLURM batch jobs were used to execute tests for each configuration (Bare-metal, Podman, Apptainer, Charliecloud) individually.

A bash script was executed to measure the nodes CPU and memory utilization over time. These scripts were deployed via SSH to minimize overhead.

4 Results and Analysis

The following section lists the results of all benchmark execution.

4.1 OSU Micro-Benchmarks

OSU! (**OSU!**) benchmarks Point-to-point latency, unidirectional bandwidth, bidirectional bandwidth, and MPI_Allreduce latency were measured across all four environments. Results are summarized below.

4.1.1 Point-to-Point Latency (osu_latency):

Latency remained nearly identical across all configurations for message sizes from 1 byte to 4 MiB. Differences were within 2–3%, confirming that the external MPI launch strategy introduces no persistent per-message overhead. The complete results are given in the Table 3. And the latency versus message size curves are shown in Figure 5.

Table 3: OSU MPI Point-to-Point Latency (ms).

Size (Bytes)	**SLURM**	**Podman**	**Apptainer**	**Charliecloud**
1	0.13616	0.13887	0.13552	0.13877
2	0.14235	0.14445	0.14094	0.14262
4	0.14366	0.13844	0.14129	0.1427
8	0.14298	0.14025	0.13664	0.14241
16	0.14805	0.14039	0.13917	0.13744
32	0.14097	0.14107	0.14036	0.13812
64	0.14881	0.14314	0.13665	0.14076
128	0.14841	0.14494	0.13942	0.14004
256	0.14714	0.13955	0.14274	0.14581
512	0.14899	0.13708	0.14412	0.13796
1024	0.1454	0.13799	0.14859	0.14274
2048	0.23906	0.23349	0.23537	0.23429
4096	0.2435	0.24018	0.24405	0.24082
8192	0.29443	0.29444	0.29484	0.29412
16384	0.71614	0.69229	0.67546	0.69539
32768	1.36927	1.37407	1.31763	1.31124
65536	2.61201	2.63106	2.62847	2.70448
131072	5.25319	5.23053	5.34373	5.33011
262144	10.44652	10.46841	10.61041	10.43793
524288	21.00324	21.04763	21.11573	21.04601
1048576	42.09475	42.03737	42.10999	42.1314
2097152	84.1279	84.15076	84.16437	84.12554
4194304	168.23026	168.25461	168.33817	168.31264

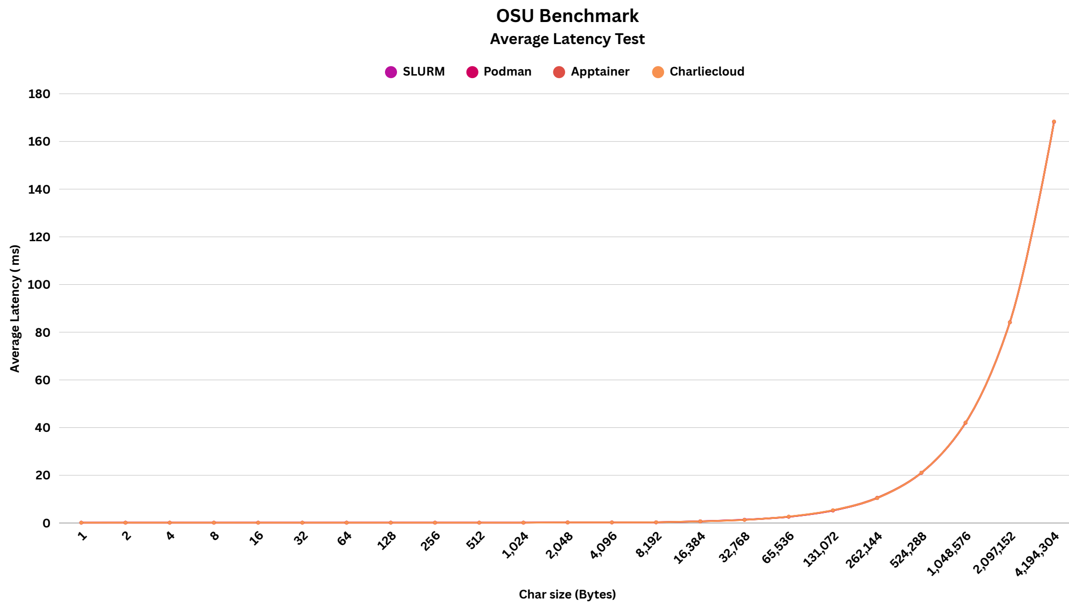

Figure 5: MPI latency (ms) vs. message size (bytes) for all four execution environments.

4.1.2 Point-to-Point Bandwidth (osu_bw):

Bandwidth was comparable at small message sizes (up to 256 bytes). At larger sizes (1–131 KB), bare metal consistently achieved the highest throughput. At 131 KB, bare metal outperformed Podman by 9.5%, Apptainer by 16.5%, and Charliecloud by 20.4%. These gaps persisted at larger message sizes, with Charliecloud showing the largest deviation. The results are presented in Table 4 and Figure 6.

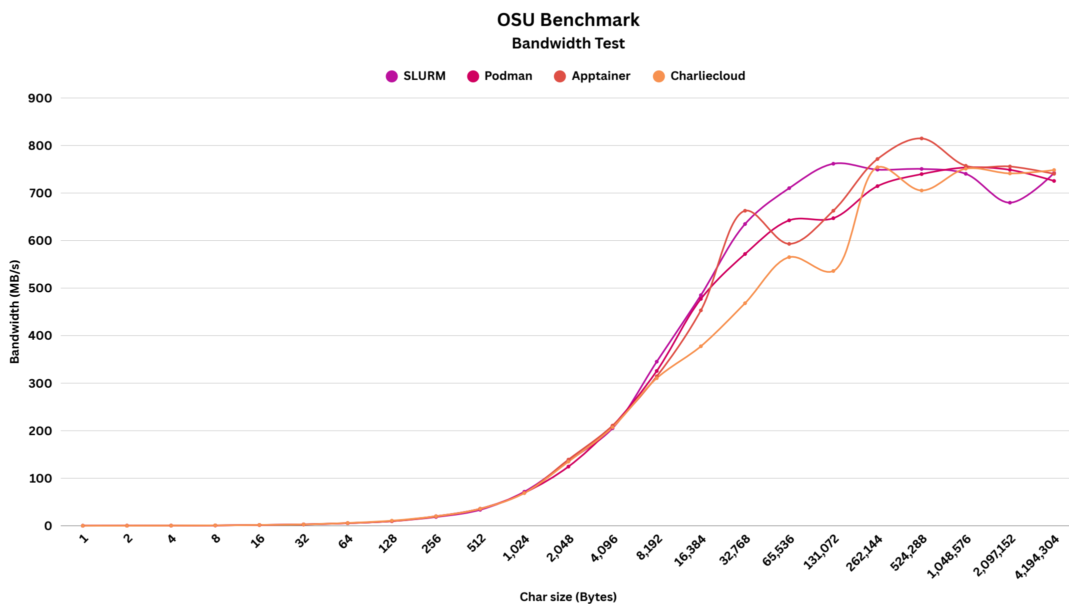

Figure 6: MPI Bandwidth (MB/s) vs. message size (bytes) for all four execution environments.

Table 4: OSU MPI Point-to-Point Bandwidth (MB/s).

Size (Bytes)	SLURM	Podman	Apptainer	Charliecloud
1	0.09	0.09	0.09	0.1
2	0.19	0.19	0.2	0.19
4	0.38	0.35	0.39	0.38
8	0.77	0.75	0.83	0.78
16	1.57	1.73	1.6	1.72
32	2.96	2.66	2.96	2.72
64	5.42	5.54	5.59	5.91
128	9.71	9.78	9.82	10.74
256	18.48	20.09	19.28	20.13
512	33.58	35.77	35.76	35.72
1024	71.57	69.43	69.06	68.85
2048	136.32	124.51	139.19	134.96
4096	205.27	209.93	210.76	207.18
8192	345.22	325.66	314.4	310.53
16384	485.11	477.27	453.19	377.64
32768	634.8	571.75	662.74	468.2
65536	710.24	642.55	593.08	565.09
131072	761.63	647.04	662.79	536
262144	749.22	714.51	771.59	754.23
524288	750.7	739.86	814.86	705.36
1048576	740.45	753.73	757.35	751.19
2097152	679.71	748.81	755.88	741.26
4194304	742.25	725.44	740.43	748.36

4.1.3 Bi-directional Bandwidth (osu_bibw):

All configurations exhibited a peak near 512 bytes, followed by a sharp drop to a stable plateau of approximately 24.9 MB/s for messages above 4 KB. Differences across runtimes remained within 1 MB/s, indicating that containerization does not affect full-duplex network performance. The results are presented in Table 5 and Figure 7.

4.1.4 Allreduce (osu_allreduce):

MPI_Allreduce latency results are presented in Table 6 and Figure 8. At small message sizes (4 KB), latency varied modestly among runtimes; bare metal showed the lowest values (e.g., 0.110 ms at 1 byte), while containers added up to 19% overhead. At larger sizes, the spread widened but remained within 10 - 20% across all configurations.

4.1.5 OSU Wall Clock Time (sec)

The following Wall clock time Table 7 is an end-to-end time for the complete OSU benchmarking suite. The total wall clock time for all runtimes is within margin of ~1 seconds.

Table 5: OSU MPI Bi-Directional Bandwidth (MB/s).

Size (Bytes)	SLURM	Podman	Apptainer	Charliecloud
1	0.22	0.21	0.23	0.23
2	0.53	0.62	0.46	0.37
4	0.9	1.37	1.23	0.92
8	2.35	2.02	2.15	1.8
16	4.02	3.7	3.67	3.44
32	7.44	7.6	7.21	8.73
64	18.02	12.79	13.08	13.8
128	28.44	29.86	27.3	31.12
256	57.3	57.88	59.67	48.8
512	102.24	98.49	113.28	101.16
1024	46.39	44.61	40.84	44.97
2048	22.11	22.35	22.63	22.71
4096	27.17	25.01	24.34	25.34
8192	23.92	25.46	24.93	25.27
16384	23.28	25.91	25.94	25.51
32768	24.19	24.5	25.78	24.11
65536	24.21	25.18	24.97	24.93
131072	24.77	24.79	24.88	24.86
262144	25	24.91	24.77	24.86
262144	24.97	25	24.9	24.97
1048576	24.9	24.92	24.93	24.91
2097152	24.94	24.94	24.95	24.93
4194304	24.91	24.93	24.94	24.92

4.2 IOR Parallel I/O Benchmarks

IOR was used to evaluate parallel I/O performance across all four execution environments. Both write and read phases were measured. The key metrics reported include bandwidth (MiB/s), IOPS, per-phase timing (open, write/read, close), and total elapsed time. A summary of all IOR results is provided in Table 8, Table 9 and Figure 9, Figure 10.

Write performance was consistent across all environments, with bandwidth within 1.2% of each other and container runtimes showing slightly higher IOPS, likely due to differences in VFS buffering. Read performance showed stark divergence: Podman achieved double the bandwidth of bare metal and more than double that of Apptainer and Charliecloud, cutting total read time in half. Apptainer and Charliecloud were nearly four times slower than bare metal in reads, with the slowdown confined to the read phase. This indicates that mount namespace implementations, rather than container overhead, critically affect I/O performance.

4.3 MDTest Metadata Benchmarks

File creation and removal rates were similar across all runtimes (230–253 ops/sec), with negligible container overhead. File stat operations showed Charliecloud leading (1.54M ops/sec, +16% vs bare metal), benefiting from its flat image format; Podman exhibited high variability (std dev 693k), possibly due to network stack interactions. File read rates

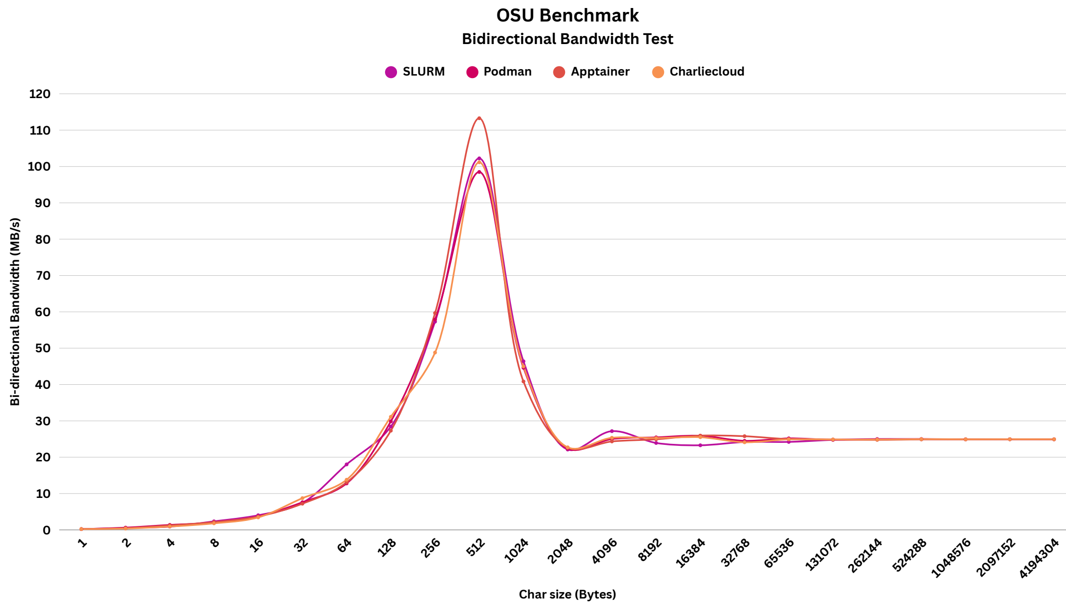

Figure 7: MPI Bandwidth (MB/s) vs. message size (bytes) for all four execution environments.

clustered around 3,100 ops/sec with minimal differences. Tree creation exposed the largest container impact: Podman and Apptainer dropped 31–44% below bare metal, while Charliecloud remained near native performance. Tree removal showed less differentiation. All the results can be found in the Table 11, Table 10,

The detailed per-runtime statistics, including the maximum, minimum, and standard deviation is as follows in Table 11.

4.4 CPU and Memory Utilization

4.4.1 OSU Benchmark CPU and Memory Utilization

During OSU benchmarks, all environments maintained a steady baseline of ~25% CPU usage (one core per node) for the entire run, with occasional spikes during collective operations and large messages. Spike heights varied: bare metal peaked at 34%, Apptainer at 35%, Charliecloud at 39–40%, and Podman at 41–42%. Podman's higher peaks reflect overhead from its conmon monitoring process and OCI namespace management, but as brief spikes they did not affect overall runtime. The sustained baseline being identical confirms that containerization adds no ongoing per-message CPU overhead for MPI workloads.

Memory usage was also tightly clustered, ranging from 449 to 508 MB across all configurations. Apptainer consistently occupied the high end (~498 MB), while Slurm and Charliecloud remained near the low end (~469 MB), with Podman in between. The narrow spread indicates that container runtimes contribute only a modest, fixed memory footprint (29–49 MB) that does not scale with communication intensity.

4.4.2 IOR and MDTest Benchmark CPU and Memory Utilization

The CPU usage behaviour during IOR test varied by environment. Bare metal and Apptainer showed short, sharp spikes of full CPU usage during writes and reads, with

Table 6: OSU Allreduce Latency (ms).

Size (Bytes)	SLURM	Podman	Apptainer	Charliecloud
1	0.11026	0.12752	0.13207	0.12312
2	0.22853	0.24533	0.25333	0.24622
4	0.24129	0.23522	0.2408	0.2525
8	0.11121	0.11989	0.11315	0.1179
16	0.12436	0.11618	0.11096	0.12067
32	0.11806	0.11487	0.11456	0.11702
64	0.11937	0.11281	0.11352	0.12403
128	0.12378	0.12059	0.11627	0.11548
256	0.11725	0.12189	0.122	0.12369
512	0.1196	0.12429	0.12749	0.1203
1024	0.11451	0.12051	0.12367	0.11224
2048	0.13124	0.13851	0.13895	0.13792
4096	0.36647	0.34291	0.35809	0.34447
8192	0.15426	0.13262	0.15518	0.14949
16384	0.35669	0.35769	0.2745	0.3377
32768	0.37367	0.42403	0.36668	0.37381
65536	4.37121	4.66375	5.29141	4.76716
131072	9.68971	7.91614	8.34021	9.31072
262144	14.95474	17.45991	18.5838	16.08293
524288	32.24011	35.17477	34.99301	33.15456
1048576	69.21687	73.3347	73.17771	74.05955

Table 7: OSU Benchmark Wall clock Time (seconds)

Runtime	Wall Clock Time (s)
SLURM	1726.403
Podman	1727.824
Apptainer	1727.818
CharlieCloud	1726.403

idle periods in between, this might meaning they spent most of the time waiting on storage. However, Podman kept the CPU busy between 50% and 100% for long stretches, with erratic swings; this extra work comes from its fuse-overlayfs layer and explains why it achieved higher throughput at the cost of more CPU. Charliecloud had multiple full-CPU bursts spread over a longer period, this matches its shorter test time but without the sustained overhead seen in Podman.

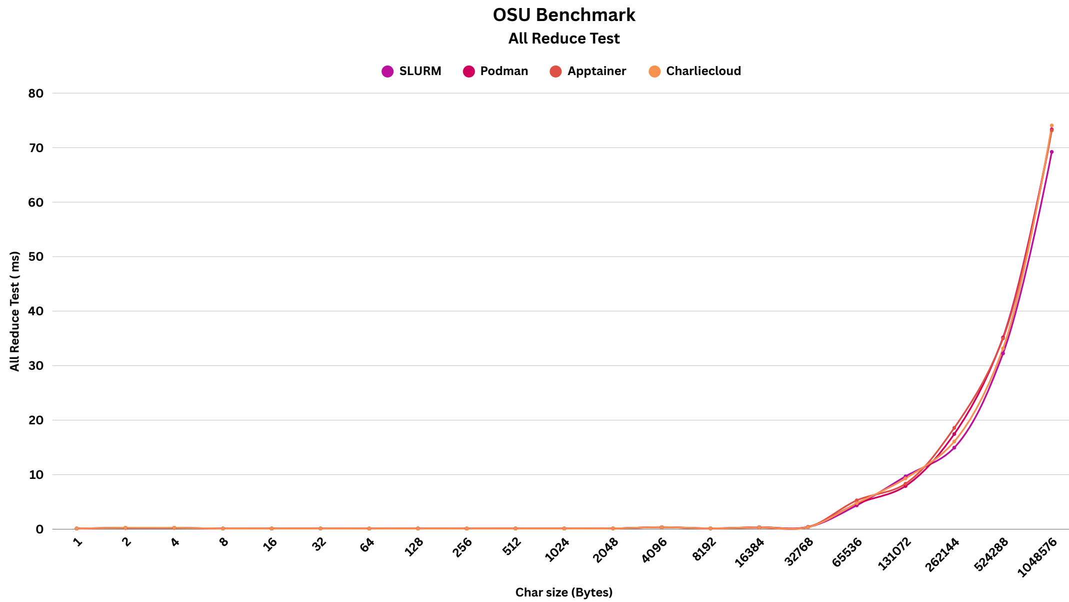

Figure 8: Allreduce latency (ms) vs message size (Bytes) for all four execution environments.

Table 8: IOR Write Test Results.

access	SLURM	Podman	Apptainer	Charliecloud
bw(MiB/s)	23.58	23.77	23.79	23.5
IOPS	2707.63	2820.91	3302.73	3522.63
Latency(s)	0.002193	0.028315	0.033431	0.02996
open(s)	0.156588	0.122379	0.014196	0.019494
wr/rd(s)	0.75638	0.726006	0.620094	0.581384
close(s)	86.27	85.69	85.54	86.64
total(s)	86.87	86.17	86.09	87.14

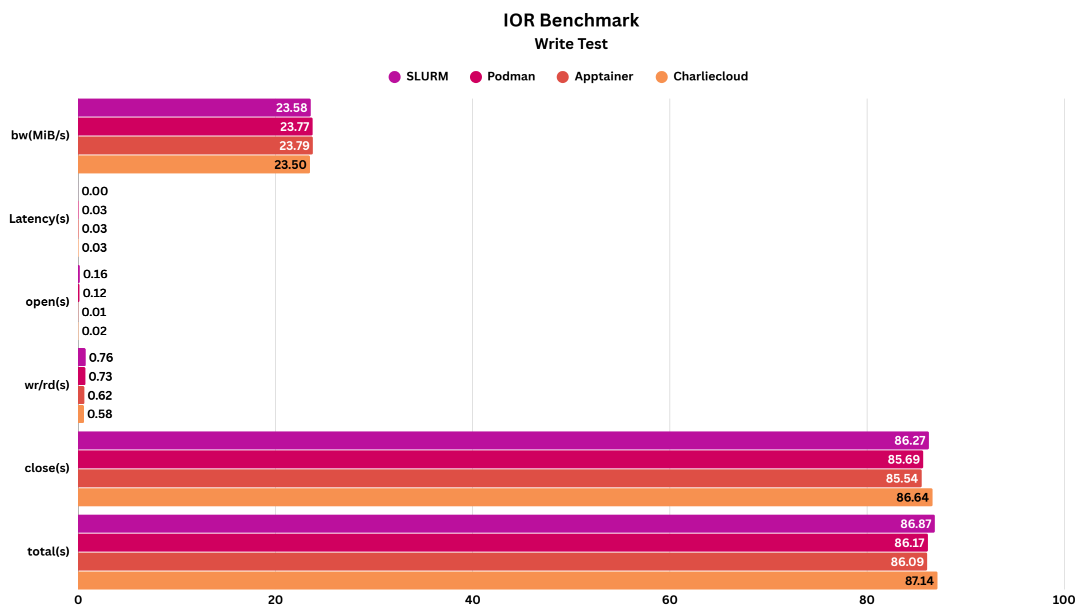

Figure 9: IOR write Results for all four execution environments.

Table 9: IOR Read Test Results.

access	SLURM	Podman	Apptainer	Charliecloud
bw(MiB/s)	11.83	23.77	11.88	11.88
IOPS	11.83	23.77	23.75	23.75
Latency(s)	0.675513	5.37	2.62	2.65
open(s)	0.001765	0.001349	0.002028	0.001838
wr/rd(s)	173.15	86.15	689.74	689.74
close(s)	4.56	86.12	22.5	15.26
total(s)	173.15	86.15	689.74	689.74

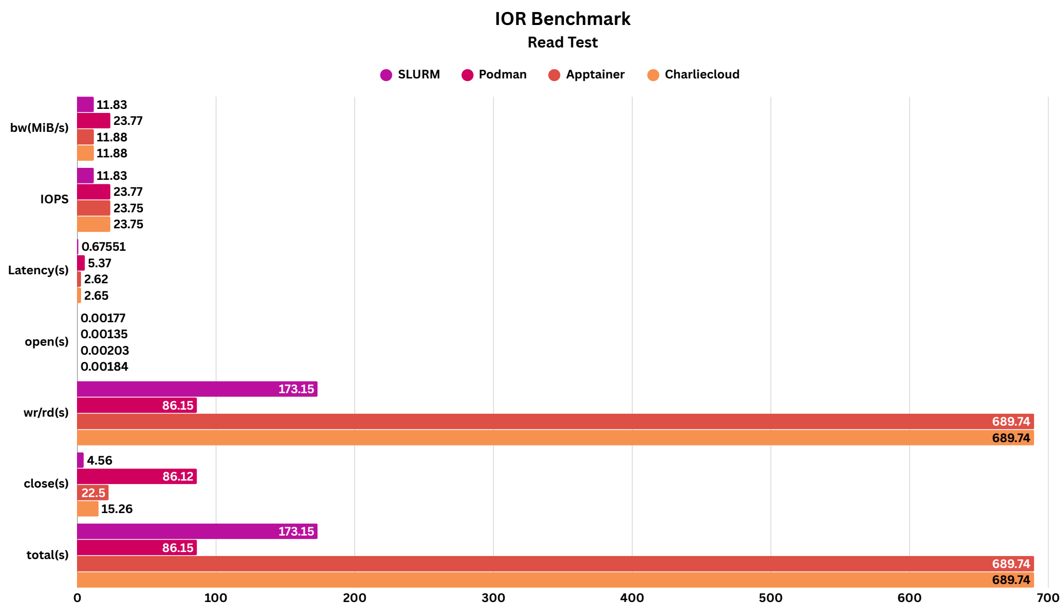

Figure 10: IOR Read Results for all four execution environments.

Table 10: MDTest Mean Metadata Operation Rates (ops/sec).

Operation	SLURM	Podman	Apptainer	Charliecloud
File Creation	250.827	247.622	253.045	229.607
File Stat	13,21,066.36	12,00,509.70	11,81,412.62	15,35,211.39
File Read	3,077.86	3,133.78	3,171.18	3,092.50
File Removal	234.55	223.194	257.347	230.189
Tree Creation	12.093	8.283	6.735	11.46
Tree Removal	7.909	8.228	9.782	7.945

Table 11: MDTest Detailed Statistics (ops/sec).

Runtime	Operation	Max	Min	Mean	Std Dev
SLURM	File Creation	265.671	225.49	250.827	22.051
Podman	File Creation	272.431	230.142	247.622	18.011
Apptainer	File Creation	291.977	230.169	253.045	27.667
Charliecloud	File Creation	253.782	204.11	229.607	20.298
SLURM	File Stat	1432909.083	1100433.95	1321066.362	191079.372
Podman	File Stat	2770270.153	558344.097	1200509.701	693501.425
Apptainer	File Stat	1338088.901	1102415.199	1181412.622	81412.811
Charliecloud	File Stat	1862710.766	1380287.493	1535211.386	113182.873
SLURM	File read	3103.157	3060.173	3077.859	22.48
Podman	File read	3227.838	3069.385	3133.78	67.315
Apptainer	File read	3246.196	3050.459	3171.179	85.679
Charliecloud	File read	3136.861	3045.935	3092.502	36.743
SLURM	File removal	251.996	221.499	234.55	15.716
Podman	File removal	244.7	211.883	223.194	15.212
Apptainer	File removal	274.361	224.347	257.347	23.338
Charliecloud	File removal	243.796	203.105	230.189	19.15
SLURM	Tree creation	15.493	8.681	12.093	3.406
Podman	Tree creation	8.998	6.898	8.283	0.98
Apptainer	Tree creation	7.679	5.579	6.735	0.87
Charliecloud	Tree creation	17.875	6.042	11.46	4.882
SLURM	Tree removal	13.33	4.279	7.909	4.784
Podman	Tree removal	13.919	4.807	8.228	4.052
Apptainer	Tree removal	13.381	4.539	9.782	3.793
Charliecloud	Tree removal	9.139	5.876	7.945	1.469

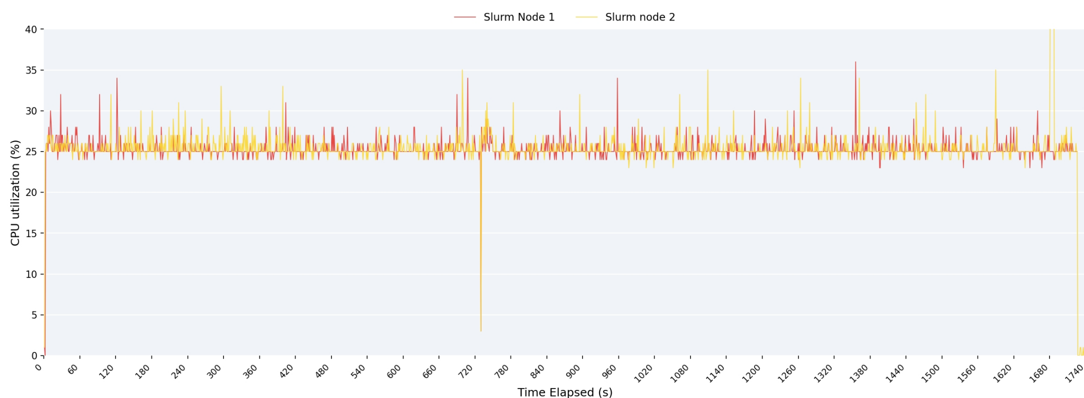

Figure 11: OSU SLURM CPU Utilization

Figure 12: OSU Podman CPU Utilization

Figure 13: OSU Apptainer CPU Utilization

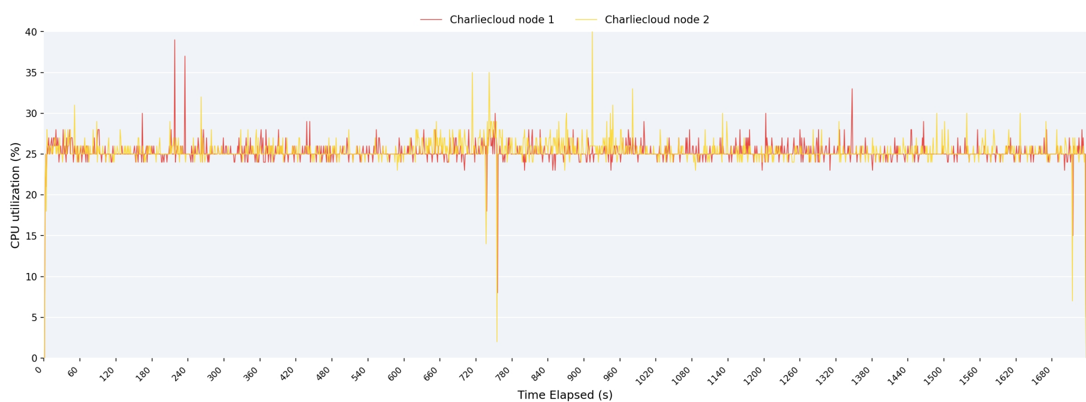

Figure 14: OSU CharlieCloud CPU Utilization

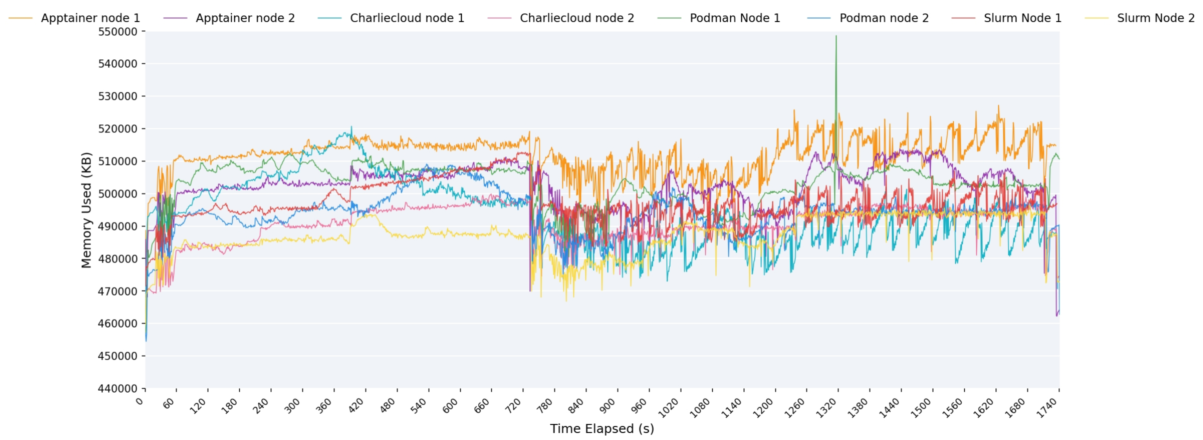

Figure 15: OSU all runtime Memory Utilization

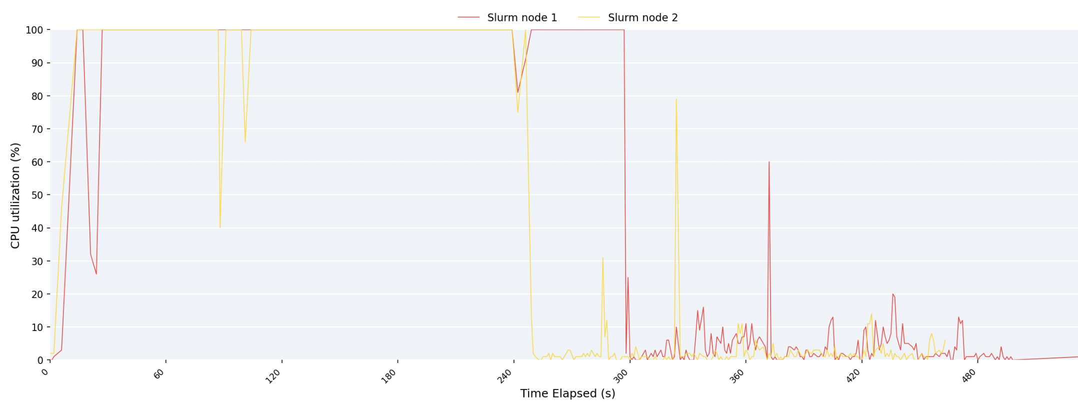

Figure 16: IOR and MDTest SLURM CPU Utilization

Figure 17: IOR and MDTest Podman CPU Utilization

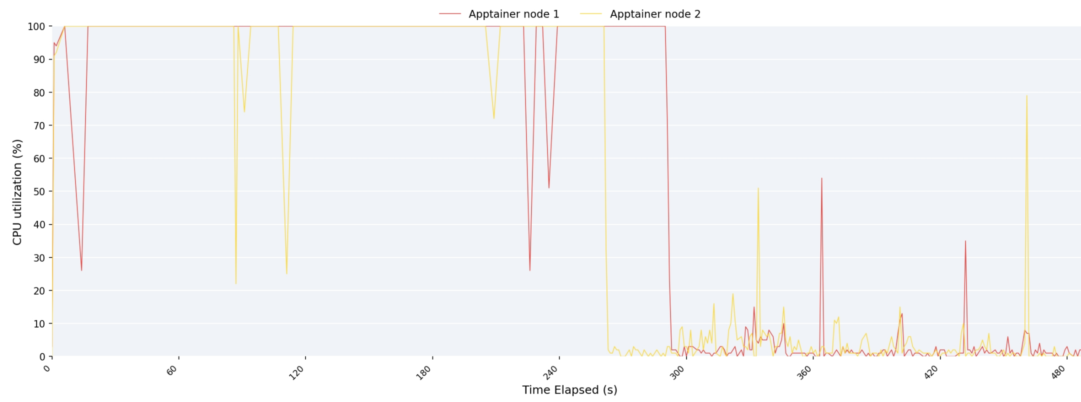

Figure 18: IOR and MDTest Apptainer CPU Utilization

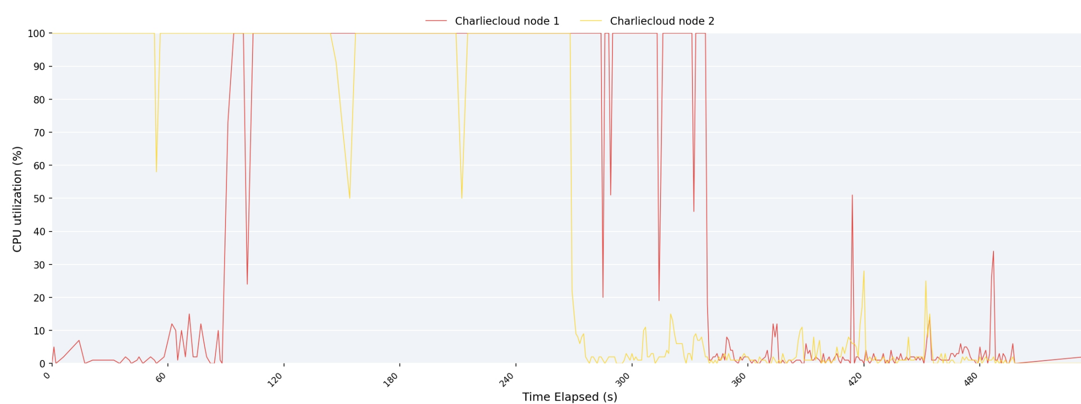

Figure 19: IOR and MDTest CharlieCloud CPU Utilization

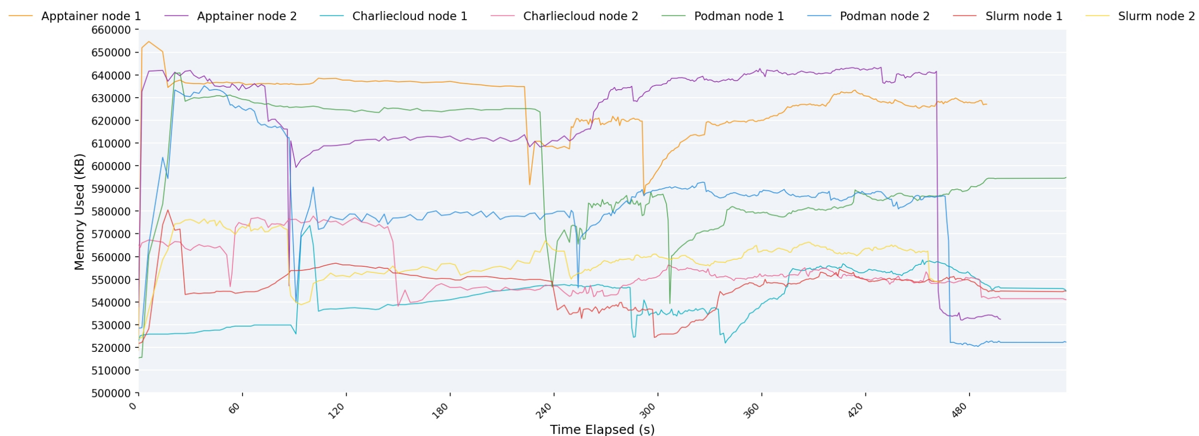

Figure 20: IOR and MDTest all Memory Utilization

Memory usage during IOR showed clear divergence across runtimes. Apptainer exhibited the highest footprint, climbing to 625–644 MB and staying elevated 98–117 MB above bare metal (508–537 MB). This is attributed to its SquashFS mounting mechanism and in-kernel decompression buffers that persist throughout I/O workloads. Podman occupied a middle range (566–625 MB), reflecting overhead from fuse-overlayfs and runtime infrastructure, while Charliecloud was slightly lower, consistent with its lighter design. The elevated memory in Apptainer occurred despite CPU profiles nearly identical to bare metal, highlighting a trade-off relevant for memory-constrained HPC nodes where multiple jobs share resources.

5 Discussion

The benchmark results reveal a nuanced picture that cannot be reduced to a simple characterization of container overhead. Across the three evaluated dimensions; MPI communication, parallel I/O, and metadata operations, the performance varied substantially by runtime and workload type.

5.1 MPI communication

All three runtimes delivered near-native networking performance. Latency differences were under 2% across the board. Charliecloud matched bare-metal performance at every tested message size. At large message sizes, bandwidth converged across all environments; at intermediate sizes, Apptainer and Podman exceeded bare metal marginally, likely attributable to MPI send buffer optimizations.

5.2 I/O and metadata

Write speeds were consistent everywhere around 23.5–23.8 MiB/s and container IOPS sometimes beat bare metal (Charliecloud was +30%, Apptainer +22%). Read performance was the most variable dimension: Podman delivered twice the read bandwidth of bare metal, while Apptainer and Charliecloud required approximately four times longer to complete the same read workload.

Podman's FUSE-based overlay filesystem may improve read performance by leveraging aggressive kernel page caching, whereas Apptainer's SquashFS mount requires on-demand decompression, increasing latency.

Metadata performance was mixed. Charliecloud was great at file stats (+16% over bare metal). Apptainer was best at file removal. But tree creation (making deep directories) took a big hit with Apptainer (–44%) and Podman (–31%).

5.3 CPU and memory

For MPI workloads, CPU usage was identical across all runtimes (steady ~25% per node). For I/O, Podman consumed sustained CPU (50–100%) due to its FUSE layer, while Apptainer's CPU pattern mirrored bare metal. Memory overhead was runtime-specific: Apptainer used ~120 MB more than bare metal during I/O (SquashFS buffers), Podman added 60–80 MB, and Charliecloud had the smallest footprint.

5.4 Runtime Selection Guidelines

- **Apptainer:** Great for compute-heavy MPI jobs; fastest overall IOR time (2.6% faster than bare metal). But reads suffer badly, and tree creation is slow.
- **Charliecloud:** The most transparent for MPI; exact bare-metal network performance, fastest file stats, and low memory overhead. Its read slowdown is the same as Apptainer's.
- **Podman:** The read champion (2× bare-metal bandwidth) but at the cost of higher CPU and inconsistent metadata performance. If your workload is read-intensive, it's worth a look.

5.5 Limitation

The generalizability of these results is constrained by the use of virtualized nodes with Gigabit Ethernet (not high-speed HPC interconnects), a small cluster size (two compute nodes), and micro-benchmarks instead of real-world applications. Performance on larger clusters with InfiniBand or production workloads may differ.

6 Conclusion

This study compared Apptainer, Podman, and Charliecloud against a bare-metal baseline on a three-node SLURM cluster, evaluating MPI communication, parallel I/O, metadata performance, and resource usage.

6.1 Key Findings

MPI: All runtimes achieved near-native performance; latency differences <2%, bandwidth convergence at large messages.

I/O: Write throughput consistent (~23.5 MiB/s); containers showed higher IOPS. Read performance diverged sharply: Podman doubled bare-metal bandwidth; Apptainer and Charliecloud were four times slower.

Metadata: Charliecloud led file stats; tree creation heavily penalized Apptainer (~44%) and Podman (~31%).

Resource usage: MPI CPU overhead negligible. During I/O, Podman consumed sustained high CPU (FUSE), Apptainer matched bare metal. Memory overhead: Apptainer ~120 MB, Podman 60–80 MB, Charliecloud lowest.

6.2 Practical Implications for HPC Practitioners

- If you're running tightly coupled MPI jobs and want the most predictable, minimal-overhead experience, Charliecloud is a great choice. It matches bare metal perfectly on MPI, has the smallest memory footprint, and just stays out of the way.
- For general-purpose HPC work where you need good Slurm integration, mature tooling, and solid overall performance, Apptainer is hard to beat. It had the fastest total runtime in our IOR tests, near-native MPI, and a strong feature set, just be mindful of read-intensive storage paths.
- If your workload is read-heavy, post-processing or data staging Podman offers a real advantage with its 2× read throughput over bare metal. Yes, it burns more CPU during I/O, but for many read-intensive jobs that trade-off may be well worth it.

There's plenty more to explore. Future work should look at larger clusters with high-speed interconnects (like InfiniBand HDR or 100 GbE), dig into the kernel-level details behind the read performance differences we saw, run real-world HPC applications (GROMACS, OpenFOAM, WRF) instead of just micro-benchmarks, and test GPU workloads with CUDA-enabled versions of these runtimes.

References

[App] Apptainer Project, Linux Foundation. *Apptainer: Application Containers for Linux*. https://apptainer.org/. Accessed: 2026-03-22.

[Can22] Canonical Ltd. *Ubuntu 22.04 LTS (Jammy Jellyfish)*. https://ubuntu.com/blog/ubuntu-22-04-lts-released. Accessed: 2026-03-22. 2022.

[Con] Containers Community. *conmon: An OCI Container Runtime Monitor*. https://github.com/containers/conmon. Accessed: 2026-03-22.

[Doc] Docker Inc. *Docker Overview*. https://docs.docker.com/get-started/docker-overview/. Accessed: 2026-03-22.

[Gab+04] Edgar Gabriel et al. "Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation". In: *Proceedings of the 11th European PVM/MPI Users' Group Meeting*. Springer, Berlin, Heidelberg, 2004, pp. 97–104. DOI: 10.1007/978-3-540-30218-6_19.

[KSB17] Gregory M. Kurtzer, Vanessa Sochat, and Michael W. Bauer. "Singularity: Scientific Containers for Mobility of Compute". In: *PLOS ONE* 12.5 (2017), e0177459. DOI: 10.1371/journal.pone.0177459.

[Law] Lawrence Livermore National Laboratory and Contributors. *IOR and MDTest: Parallel Filesystem I/O Benchmark*. https://github.com/hpc/ior. Accessed: 2026-03-22.

[MPI23] MPI Forum. *MPI: A Message-Passing Interface Standard Version 4.1*. Tech. rep. Accessed: 2026-03-22. MPI Forum, Nov. 2023.

[Net] Network-Based Computing Laboratory, The Ohio State University. *OSU Micro-Benchmarks v7.4*. https://mvapich.cse.ohio-state.edu/benchmarks/. Accessed: 2026-03-22.

[Ope] Open Container Initiative. *Open Container Initiative — Image Format Specification*. https://opencontainers.org/. Accessed: 2026-03-22.

[Pod] Podman Project. *Podman Documentation*. https://docs.podman.io/en/latest/. Accessed: 2026-03-22.

[PR17] Reid Priedhorsky and Tim Randles. "Charliecloud: Unprivileged Containers for User-Defined Software Stacks in HPC". In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC'17)*. ACM, 2017. DOI: 10.1145/3126908.3126925.

[Sch] SchedMD LLC. *SLURM Workload Manager Documentation*. https://slurm.schedmd.com/documentation.html. Accessed: 2026-03-22.

[Sud] Akihiro Suda. *slirp4netns: User-mode Networking for Unprivileged Network Namespaces*. https://github.com/rootless-containers/slirp4netns. Accessed: 2026-03-22.

[The] The Linux Kernel Authors. *Overlay Filesystem — The Linux Kernel Documentation*. https://www.kernel.org/doc/html/latest/filesystems/overlayfs.html. Accessed: 2026-03-22.

[YJG03] Andy B. Yoo, Morris A. Jette, and Mark Grondona. “SLURM: Simple Linux Utility for Resource Management”. In: *Job Scheduling Strategies for Parallel Processing (JSSPP 2003)*. Vol. 2862. Lecture Notes in Computer Science. Springer, Berlin, Heidelberg, 2003, pp. 44–60. DOI: 10.1007/10968987_3.

A Code samples

The source code for the project can be found on GitHub: https://github.com/pathakutkarsh/container-runtime-testing-slurm/tree/main