On the Use of Containers in High Performance Computing Environments

Subil Abraham†, Arnab K. Paul†, Redwan Ibne Seraj Khan, Ali R. Butt
Virginia Tech
{*subil, akpaul, redwan, butta*}*@vt.edu*

***Abstract*—The lightweight nature, application portability, and deployment flexibility of containers is driving their widespread adoption in cloud solutions. Data analysis and deep learning (DL)/machine learning (ML) applications have especially benefited from containerization. As such data analysis is adopted in high performance computing (HPC), the need for container support in HPC has become paramount. However, containers face crucial performance and I/O challenges in HPC. One obstacle is that while there have been HPC containers, such solutions have not been thoroughly investigated, especially from the aspect of their impact on the crucial HPC I/O throughput. To this end, this paper provides a first-of-its-kind empirical analysis of state-of-the-art representative container solutions (Docker, Podman, Singularity, and Charliecloud) in HPC environments. We also explore how containers interact with an HPC parallel file system like Lustre. We present the design of an analysis framework that is deployed on all nodes in an HPC environment, and captures CPU, memory, network, and file I/O statistics from the nodes and the storage system. We are able to garner key insights from our analysis, e.g., Charliecloud outperforms other container solutions in terms of container start-up time, while Singularity and Charliecloud are equivalent in I/O throughput. But this comes at a cost, as Charliecloud invokes the most metadata and I/O operations on the underlying Lustre file system. By identifying such trade-offs and optimization opportunities, we can enhance HPC containers performance and the ML/DL applications that increasingly rely on them.**

***Keywords*-Container Performance, High Performance Computing, Parallel File Systems, HPC Storage and I/O**

I. Introduction

Containers are experiencing massive growth as the deployment unit of choice in a broad range of applications from enterprise to web services. This is especially true for leveraging the flexibility and scalability of the cloud environments [1]. Extant data analysis including deep learning (DL) and machine learning (ML) are further driving the adoption of containerized solutions. Containers offer highly desirable features: they are lightweight, comprehensively capture dependencies into easy-to-deploy images, provide application portability, and can be scaled to meet application demands. Thus, containers free the application developers from lower-level deployment and management issues, allowing the developers to focus on their applications and in turn significantly reduce the time-to-solution for mission-critical applications.

† Made equal contribution to this work.

Modern data analysis is compute and I/O hungry [2], [3] and require massive computing power, far beyond the capabilities of what a scaled-up single node can provide. At the same time, more and more scientific workflows rely on DL/ML to analyze and infer from scientific observations and simulations data. Thus, there is a natural need for data analysis and I/O-intensive DL/ML support in HPC. A challenge to this end is that modern DL/ML software stacks are complex and bespoke, with no two setups exactly the same. This leads to an operations nightmare of addressing library, software, and other dependencies and conflicts. To this end, enterprise data analysis solutions have adopted containers given their ability to encapsulate disparate services that can be orchestrated together, reused, and reinstantiated transparently. Thus, efficiently supporting containers in HPC systems will provide a suitable and effective substrate for the crucial data analysis tasks.

The use of containers is being explored in the HPC environment, and have produced benefits for large scale image processing, DL/ML workloads [4]–[6], and simulations [7]. Due to user demand and ease of use, containers are also becoming an integral part of HPC workflows at leadership computing facilities such as at Oak Ridge [8] and Los Alamos [9] national laboratories. Currently available HPC container solutions include Singularity [10], Podman [11], Shifter [12], and Charliecloud [13]. Among these, Singularity and Charliecloud are designed especially to make the best use of HPC resources.

In addition to meeting the computing needs of HPC applications, containers have to support the dynamic I/O requirements and scale, which introduce new challenges for data storage and access [14]–[17]. Moreover, different container solutions incur different overheads. Therefore, it is crucial to analyze the interaction of containers with the underlying HPC storage system—one of the most important HPC resources. The knowledge gained can help build an effective ecosystem where both parallel file systems (PFS) and containers can thrive. There have been several works that have examined the performance of containers on HPC systems [7], [18]–[23]. There have also been studies on parallel file systems for use in cloud environments [24]–[28]. However, to the best of our knowledge, existing works have not studied the behavior of HPC storage system, especially the parallel file system, in service of containers.

In this paper, we examine the performance of different

container solutions on a parallel file system. We present a framework to analyze different representative container solutions running atop HPC storage. The framework collects and compares performance metrics from both client and storage nodes, including CPU, memory, network, and file system usage. We also collect metadata and I/O statistics gathered from the PFS. We focus on file I/O, specifically analyzing the effects of I/O operations made by containers. We use the containerized image of Sysbench [29] to run CPU, memory, and file I/O benchmarks. We also use real-world representative HPC workloads such as HaccIO [30] and IOR [31] to explore the I/O impact of the studied containers.

We study four extant container solutions: Docker, Podman, Singularity, and Charliecloud. The HPC PFS that we selected for our analysis is Lustre [32], as it is the most widely used file system in HPC [33]. According to the latest Top 500 list [34], Lustre powers ∼60% of the top 100 supercomputers in the world, and will power Frontier [35], the world's first exascale supercomputer. While the analysis is based on the experiments on the Lustre file system, the results can be easily extended to other HPC file systems. Our analysis shows that Charliecloud gives better performance in terms of container startup time. Both Singularity and Charliecloud give similar I/O throughput on the Lustre file system. However, when analyzing the number of requests that arrive at Lustre's metadata and object storage servers, Charliecloud fares the worst. Our aim is to provide insights to help both HPC practitioners and system administrators so that they can obtain best I/O performance when running HPC container solutions.

II. Background

In this section, we give an introduction to the container technology, and describe some of the commonly used container solutions. We also describe HPC parallel file systems, especially the architecture of Lustre file system.

A. Containers

Containers differ from other virtualization methods like KVM [36] that explicitly virtualize the hardware and run separate kernels. In contrast, containers on the same node share the same kernel. Containers are able to provide a VM like isolation through the use of cgroups [37] and namespaces [38]. As a result, a container can provide the necessary isolation between applications without the heavy resource utilization of a virtual machine, in turn allowing many more containers than VMs to run simultaneously on a node.

Containers provide the ability to read and write data (I/O) by *bind-mount*ing a directory in the container to a directory on the node that hosts the container. Therefore, container I/O to a directory actually takes place on the underlying host bind-mounted directory.

Beyond isolation, arguably the main benefit of containerization is the ease of packaging the application and all of its dependencies in a readily deployable format. This advantage is particularly useful for HPC, where such packaging makes deployment of ML/DL significantly easier by freeing the users from installing and managing software dependencies in complex HPC systems.

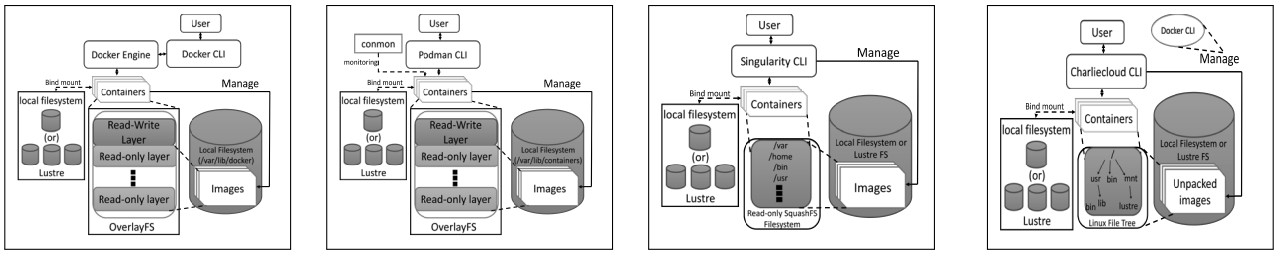
B. Containerization Solutions

1) Docker: Docker [39] is one of the most popular container solutions. Although approaches such as LXC [40] have been around for a long time, it was Docker that popularized containerization as a better and more efficient solution for providing isolation for applications, especially in the cloud computing setting. Figure 1a shows a simplified architecture overview of Docker.

However, a number of drawbacks in Docker has limited its enthusiastic adoption in HPC environments. Firstly, the approach depends on an always-on Docker daemon that the Docker client (e.g. the Command Line Interface (CLI)) interacts with to perform container operations. This daemon spawns the containers as child processes, which make it a single point of failure and can possibly lead to orphaned containers in case the daemon crashes. The entire container management ecosystem of Docker rests inside this single daemon, making it bloated and unsuitable for mission-critical HPC deployments. Docker also needs root permissions to run. This almost always contradict with the HPC center policies that disallow such permissions to avoid execution of harmful arbitrary code. In addition, Docker stores its images split up into a number of read-only layers. When a Docker container is started from an image, the daemon uses OverlayFS [41] to create a union of all of its layers and creates a read-write layer at the top. However, a parallel file system such as Lustre does not support OverlayFS. So an image must be explicitly downloaded and run from a supported local file system—a step that introduces unnecessary overhead and limits scalability. Docker also does not provide useful support for native MPI operations that are often needed for typical HPC applications.

Storing of Docker images in the Lustre file system have also been proposed [42]. However, this solution involved an inefficient way of abusing the device-mapper driver and creating loopback devices in Lustre so that the data-root can be pointed to those devices. The device-mapper driver is now deprecated and will be removed in future Docker releases, so such a solution is also not future-proof.

2) Podman: Podman [43] provides a comprehensive container solution similar to Docker. The main driving force for the development of Podman has been the need to avoid having a privileged management daemon used in Docker. Podman replaces the daemon-client architecture of Docker with individual processes that run the containers, and uses *conmon* [44] to provide the necessary monitoring and debug-



(a) Docker **(b)** Podman **(c)** Singularity **(d)** Charliecloud

Figure 1: Architecture of the studied container solutions.

ging capabilities. Figure 1b shows the overview of Podman architecture.

Podman can also run rootless containers through the use of user namespaces [45], which ensures additional security and separation among containers, especially in HPC scenarios.

Podman, however, also relies on OverlayFS to store and run containers similar to Docker. The layer structure of the Podman images is OCI compliant, but the dependence on OverlayFS is a roadblock for storing images on distributed file systems in HPC environments as discussed earlier.

3) Singularity: Singularity [10] is a container framework tailored specifically for HPC environments. Its goals are to provide isolation for workloads while preventing privilege escalation, offer native support for MPI, Infiniband, and GPUs, and support ease of portability and reusability through distributing container images as a single SquashFS [46] file. Thus, Singularity can store images and run containers from the Lustre file system. Figure 1c shows the overview of Singularity.

A key feature of Singularity is its focus on security such as root file system encryption, as well as cryptographically signing containers images. Singularity containers are read-only by default and all write operations can only be done via bind-mounted directories. These features make Singularity attractive for HPC systems.

4) Charliecloud: Another HPC-focused containerization solution is Charliecloud [13], which also provides native MPI support similar to Singularity. Charliecloud differs from Singularity in that Charliecloud focuses on simplicity of architecture instead of portability. Its primary aim is to provide a way to encapsulate dependencies with minimal overhead. Charliecloud employs Docker or other image builder tools such as Buildah [47], umoci [48], and the in-built `ch-grow` to build an image, and then extracts all the contents of that image into an unpacked file tree format. As a result, instead of existing as compressed layers (as in Docker and Podman) or as a single file (as in Singularity), Charliecloud images are stored as a file tree. Applications can be run from the file tree by *pivot_root*ing [49] into it, as shown in Figure 1d. Charliecloud also provides a means of constructing and running compressed SquashFS images akin to Singularity, which is expected to perform similar to Singularity. Unlike Docker, which isolates everything, Charliecloud only uses separate namespaces for mount and user, whereas the network and process namespaces are shared with the underlying OS. Because of this minimal isolation, Charliecloud is intended to be used in a trusted environment (similar to an HPC allocation).

Discussion: Although Docker and Podman are designed for cloud systems, and not necessarily for HPC environments, they are well-established and widely-used container solutions and their efficacy in HPC should be explored. This is further underscored by the observation that HPC-focused container solutions also support Docker images. Moreover, systems such as Charliecloud can employ Docker as needed.

We note that Shifter [12] is another prominent HPC container solution in the HPC field. We did create and evaluate an approximated setup of Shifter from available resources. However, Shifter's close dependence on difficult-to-recreate NERSC's supercomputers and their job scheduling setup, the results from our Shifter study did not yield useful insights for general HPC containers. Therefore, we do not include these observations in this paper.

C. HPC Parallel File Systems

HPC PFSs are designed to distribute file data across multiple servers so that multiple clients can access a file system simultaneously at extreme scales. Typically, HPC PFS consists of *clients* that read or write data to the file system, *data servers* that store the data, *metadata servers* that manage the metadata and placement of data on the data servers, and networks to connect these components. Data may be distributed (divided into stripes) across multiple data servers to enable parallel reads and writes. This level of parallelism is transparent to the clients, which access the PFS similar to accessing a local file system. Therefore, important functions of a PFS include avoiding potential conflict among multiple clients and ensuring data integrity and system redundancy. The most common HPC PFS include Lustre, GlusterFS, BeeGFS, and IBM Spectrum Scale, with Lustre being one of the most extensively used in HPC environments.

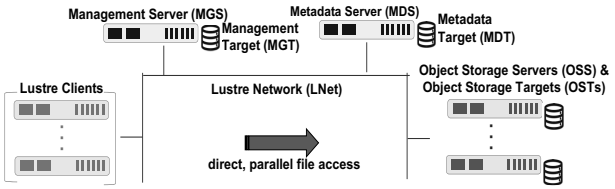


Figure 2: An overview of Lustre architecture.

1) Lustre File System: Figure 2 shows the high-level overview of Lustre architecture. Lustre has a client-server network architecture and is designed for high performance and scalability. The *Management Server (MGS)* is responsible for storing the configuration information for the entire Lustre file system. This persistent information is stored on the *Management Target (MGT)*. The *Metadata Server (MDS)* manages all the namespace operations for the file system. The namespace metadata, such as directories, file names, file layout, and access permissions are stored in a *Metadata Target (MDT)*. Every Lustre file system must have a minimum of one MDT. *Object Storage Servers (OSSs)* provide the storage for the file contents. Each file is stored on one or more *Object Storage Target (OST)*s mounted on the OSS. Applications access the file system data via *Lustre clients* that interact with OSSs directly for parallel file accesses. The internal high-speed data networking protocol for the Lustre file system is abstracted and is managed by the *Lustre Network (LNet)* layer.

III. Analysis Framework

A. Lustre Deployment

We use a Lustre cluster of ten nodes with one MDS, seven OSSs and two clients for our study. Each node runs CentOS 7 with an AMD FX-8320E eight-core 3.2 GHz processor, 16 GB of RAM, and a 512 GB SSD. All nodes are interconnected with 10 Gbps Ethernet. Furthermore, each OSS has five OSTs, each supporting 10 GB of attached storage. The setup offers a 350 GB Lustre store for our analysis.

B. Container Deployment

For deploying Docker and Podman on our HPC setup, we utilized the clients' local storage because Lustre does not support OverlayFS (as discussed in Section II-B). For Singularity and Charliecloud, the analysis was mainly done with Lustre. However, we used both Lustre and clients' local storage for startup time tests and the final multi-node throughput tests. We benchmark CPU, memory, and file I/O performance of the containers with Sysbench [29]. We also studied Lustre I/O throughput under containerized real-world HPC workloads including HaccIO [30] and IOR [31].

C. Analysis Framework

We build a framework, seen in Figure 3, to encapsulate all the required operations of building the container images, running them, and collecting and analyzing the metrics.

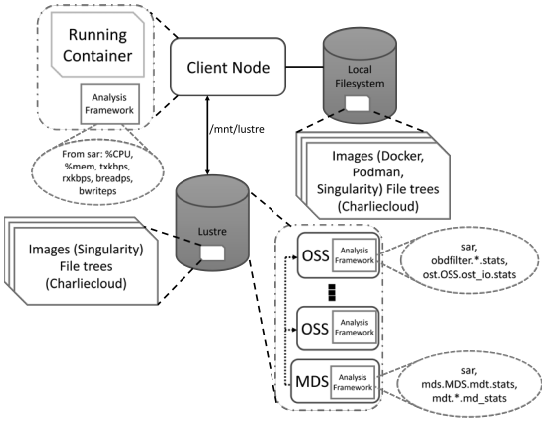


Figure 3: Architecture of our analysis framework.

The framework takes as input the container solution to analyze and the type of benchmark to run. When the framework is first deployed, it invokes the container image build process. The built images are stored on the local file system or on Lustre, based on the container deployment. The framework incorporates the collection of `sar` [50] metrics on the client for the benchmarks. On the Lustre server nodes (MDS and OSS), the framework collects the `sar` metrics, as well as OSS and MDS specific statistics from the individual nodes. The framework collects metrics on CPU and memory utilization, network usage, and reads and writes on the storage devices.

When the analysis framework is activated, it executes the needed servers on the Lustre nodes that will listen for GET requests from the client. The GET requests are sent from the client nodes at the start and stop of each benchmark run to trigger the start and stop of metrics collection on the Lustre nodes, respectively. After activating metrics collection, the container with the benchmark code is started and the benchmark is run. At the end of each run of the benchmark, the used container is removed from memory and disk, and the caches are cleared. For each new benchmark run, a new container is spawned from the corresponding image to avoid reuse of any cached data or old paused containers to avoid data pollution. All Sysbench benchmarks are cut off at 30 seconds (selected based on our observation of the typical benchmark runs on our target cluster) and are forcibly stopped if they do not complete in this time period. However, the HaccIO and IOR benchmarks are run to completion. The metrics collected from each of the nodes for each run are all gathered at a central location for analysis.

IV. Analysis

A. Workloads Description

Our experimental setup comprises a Lustre file system deployment of seven OSSs, and one MDS. Container operations were done through two client nodes to validate both single and multi-node setups. Our single-node workloads

employed Sysbench to benchmark memory performance, CPU performance, and sequential and random file I/O throughput. All Sysbench benchmarks are run with eight threads and measurements are averaged across ten runs.

The CPU workload from Sysbench repeatedly calculates prime numbers up to 1,000,000 to stress load the CPU until the specified timeout is reached. It then reports the number of repetitions broken down by thread during the benchmark run, and the average latency for each repetition.

The memory workload consists of repeatedly allocating a 1K block, performing sequential/random reads/writes (depending on the options passed to Sysbench), and deallocating the block. This is repeated until the timeout is reached or 100 GB of data is read or written. The benchmark then reports the amount of data read or written, and the total and per thread average time taken to complete the benchmark.

The file I/O benchmark consists of preparing 20 GB of data split across 128 files filled with random data, and then performing the sequential or random synchronous file I/O operations on the file depending on the particular file I/O benchmark being run. Sysbench reports the amount of data read and written during the benchmark run.

We use `sar` to collect system statistics during the Sysbench benchmark runs.

In addition, we also tested a real-world multi-node setup using HaccIO [30] and IOR [31] running simultaneously from two different client nodes. The HaccIO benchmark simulates the I/O patterns of the Hardware Assisted Cosmology Code [51], simulating a typical workload for HPC systems. It is run with a number of particles set to 4,096,000 for all the runs. In the IOR benchmark, each MPI process writes a block of 256 MB of data in parallel and reads it back with the processes shifted so that each process is not reading the same block it wrote. We look at the I/O throughput reported by these benchmark suites as well as the statistics reported by Lustre to study relative performance.

B. Research Questions

In analyzing and comparing the studied container solutions, we try to answer the following research questions:

Q1. How would performance compare among different container solutions running in an HPC environment?

Q2. How would the Lustre file system performance differ for different container solutions?

C. Container Startup Time

Figure 4 shows the time to startup for the studied container solutions. The higher startup time in Docker and Podman is due to the overhead of building the container from the multiple image layers, setting up the read-write layer, and setting up the monitoring of the containers. Charliecloud and Singularity do not use layers to build their containers, nor do they setup a read-write layer by default. In addition, they usually expect the job schedulers to handle the monitoring. As a result, their startup times are lower compared to that of Docker.

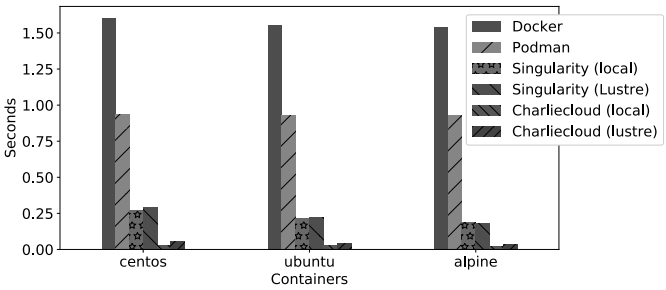


Figure 4: OS image startup times under the studied container solutions.

D. Observations on The Client

1) Resource usage spikes in container startup: Starting CPU utilization is higher for Docker and Podman as seen in Figure 5a. Docker and Podman uses additional computational resources for spawning containers and starting up monitoring, and thus have additional CPU overhead at startup. This is likely because of the additional work from starting containers from multi-layered images, which also explains their longer startup time.

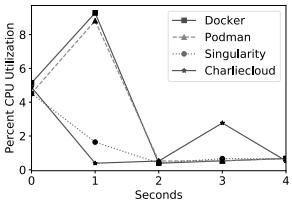
There is also a noticeable spike in data received over network for Singularity as seen in Figure 5b. This spike is because the Singularity image is stored as a single file on Lustre. This results in Singularity having to load a huge amount of data (and not just the startup portion) from the image at startup time, which in turn slows the startup process.

Figure 5c shows that Charliecloud has a spike in the data transmitted from the client node over the network. This is likely because Charliecloud has to make a larger number of MDS and OSS requests since it is a bare tree structure, not shared layers or a single file, and thus will need to access a lot of individual files from the image tree from Lustre from the outset.

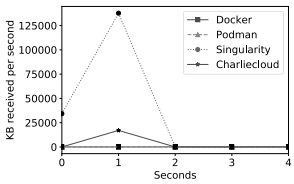
Figure 5d shows a spike in blocks written for Docker, which is likely caused by the creation of the read-write layer when starting the container. This spike for Docker is significantly higher than that for Podman, which also does the same thing. This indicates that Podman uses far fewer resources for the startup process.

2) CPU and Memory Utilization: Figures 6 and 7 show box plots of the CPU and memory utilization, respectively, for the CPU (sbcpu), memory (sbmem), and I/O (sbfileio) workloads across 10 runs. All four container solutions behave fairly similarly in how they use CPU and memory for the studied workloads.

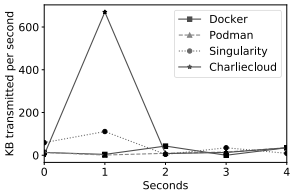
3) File I/O Throughput: Table I shows the average read and write throughput on the client node for Sysbench.



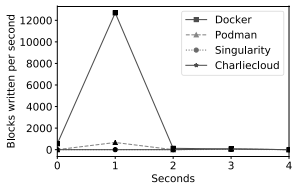
(a) Observed spike in CPU utilization for Docker and Podman during Sysbench sequential read-write.



(b) Observed spike in data received over network for Singularity during Sysbench memory random read[1].



(c) Observed spike in data transmitted over network for Charliecloud during Sysbench memory random read[1].



(d) Observed spike in blocks written for Docker during Sysbench sequential read-write.

Figure 5: Resource usage spikes observed during container startup for the Sysbench benchmarks.

Benchmarks	Docker		Podman		Singularity		Charliecloud	
	read	write	read	write	read	write	read	write
Sequential Read	226.691	0	224.137	0	222.045	0	226.13	0
Sequential Write	0	87.447	0	86.808	0	90.655	0	90.447
Sequential Read-Write	0	89.387	0	89.044	0	91.498	0	94.357
Random Read	124.404	0	128.498	0	131.483	0	123.033	0
Random Write	0	39.029	0	38.76	0	39.787	0	39.949
Random Read-Write	29.012	19.339	28.515	19.007	29.605	19.733	29.198	19.465

Table I: Throughput (MB/s) for container solutions for File I/O workloads.

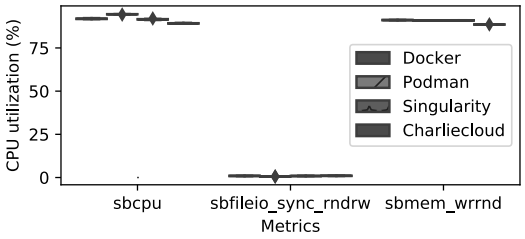


Figure 6: CPU utilization for different workloads.

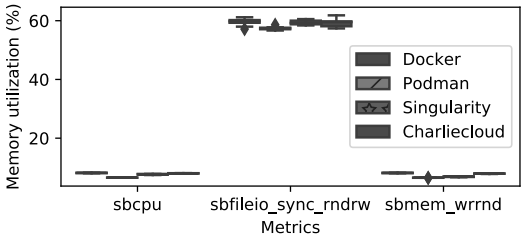


Figure 7: Memory utilization for different workloads.

Singularity and Charliecloud slightly outperform Docker and Podman in read and write throughput in most cases. Singularity performs worst for sequential read, and Charliecloud gives the worst performance for random read. But the performance differences are fairly small and do not seem to indicate significant container overhead during the benchmark run. This observation holds true for performance of real world benchmarks shown in Section IV-F even though there are observed differences in the performance of the underlying Lustre PFS .

[1]The memory read benchmark itself does not do any file or network I/O.

E. Behavior of Lustre File System

Average	Docker	Podman	Sing	Char
read_bytes/OSS (MB)	317.8	314.71	370.52	352.72
write_bytes/OSS (MB)	83.26	81.86	84.97	83.85
# of requests/OSS	9920	9760	10179	10072
req_waittime/OSS (μs)	36.59	36.55	36.06	36.37
# of requests in MDS	120768	118764	123235	121772
req_waittime in MDS (μs)	24.87	24.91	24.89	24.45
open calls in MDS	128	128	131	200

Table II: Average Lustre activity per run of the Sysbench random read/write file I/O benchmark (Sing - Singularity, Char - Charliecloud).

Table II shows the Lustre statistics for the Sysbench Random Read-Write file I/O benchmark. Singularity has the highest amount of data read per OSS. This is mainly due to the fact that Singularity has to read larger blocks of the contents from its container image from Lustre, in addition to the read operation of the benchmark itself. Docker and Podman read container images from the client's local file system. So they do not exhibit such an overhead. Singularity reads larger blocks compared to Charliecloud. This is because Singularity's container image is a large single file and would load more data at a time, whereas Charliecloud reads smaller amounts of data since Charliecloud has to only read individual files from its unpacked file tree.

For the same reasons, Singularity also shows a higher average number of requests on the MDS and OSSs. However, this is in contrast to the multi-node real world scenario where Charliecloud has comparatively higher number of OSS and MDS requests as discussed in Section IV-F. In either case, the higher number of metadata and data operations because of container overhead could potentially

lead to more metadata and I/O contention on the MDS and OSSs when working with a large number of I/O intensive containerized applications in an HPC environment.

F. Real-World Benchmarks: HaccIO and IOR

As a final step, we also obtain the I/O metrics across ten HaccIO and IOR runs on the Lustre file system run through each container solution in a multi-node setup. We run HaccIO and IOR separately on each client node to mimic a heterogeneous workload on Lustre. The number of MPI processes are varied to see how the throughput changes with increased parallelism in the workloads. In the workload, both HaccIO and IOR are run simultaneously in the multi-node setup with two, four, and eight MPI processes each.

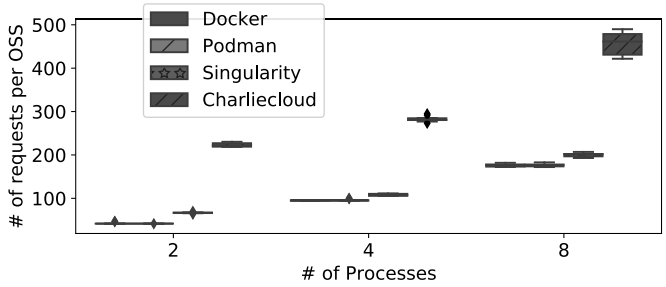


Figure 8: Box plot of number of requests per OSS across ten runs, averaged across seven OSSs, for HaccIO and IOR benchmark with different number of MPI processes.

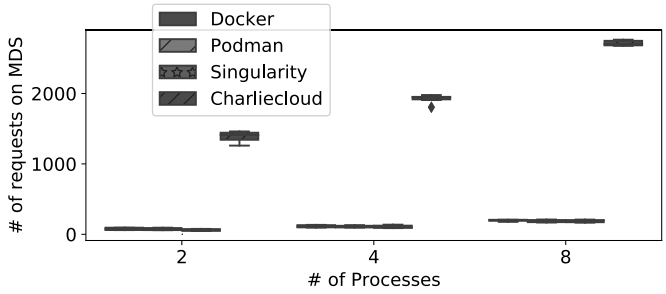


Figure 9: Box plot of number of requests across ten runs in MDS for HaccIO and IOR benchmark with different number of MPI processes.

Figures 8 and 9 show that Charliecloud, with increasing numbers of processes, has the highest amount of activity on OSSs and MDS. This is significantly higher than any other container solution, due to the unpacked file tree structure of its container image. This means that Charliecloud has to read multiple small files individually. These reads multiply as we increase the number of MPI processes, since they are all independent of each other. As a result, Charliecloud has to do a lot of reads in addition to just the work of IOR, leading to a higher number of requests on the Lustre metadata and storage server nodes. A Singularity image on the other hand is stored as just one file, and does not incur such an overhead. Therefore, Singularity scales better with increasing parallelism. For a single process, Charliecloud fares better, as it makes fewer requests to the OSSs and MDS for image data reads compared to Singularity. This is seen in Table II, which shows that Charliecloud is more efficient than Singularity in Lustre operations, though their read and write throughput are similar as seen in Table I. However, such is not the case in a multi-process setup, where Singularity behaves better from the Lustre perspective.

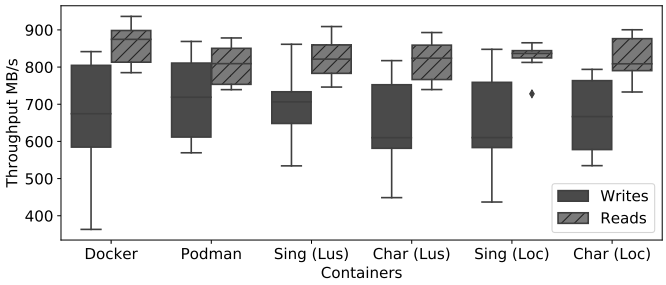


Figure 10: Box plot showing read and write throughput for IOR during multi-node run. (Sing: Singularity, Char: Charliecloud, Lus: image stored on Lustre, Loc: image stored on client's local storage).

We also performed another test of HaccIO and IOR in the multi-node setup by increasing the number of particles in HaccIO to 8,192,000 and changing the number of 256M size blocks written by each MPI process from one to four. This means that each run has eight MPI processes and the total IOR data written and read is 8 GB. This experiment allows a better look at the I/O throughput. Additionally, we perform a run of Singularity and Charliecloud from the local storage, similar to the Docker and Podman, to help see if there is difference in throughput between running the same container from local storage and PFS. We focus on the IOR read and write throughput results. Figure 10 shows that Singularity and Charliecloud do not exhibit any differences in their throughput when we compare storing the container image on Lustre versus the local file system. This indicates that running these container images from a distributed file system instead of a local file system does not affect their observed I/O performance.

G. Discussion

In answering our first research question of *how the performance of the studied container solutions differ*, the most important observation is the significant difference in startup times among the studied container solutions, as well as the spikes seen in the various metrics during container startup. Here, Charliecloud offers the best startup performance. Besides startup, the container solutions do not show any major differences in non I/O related metrics. To gauge I/O performance, it is clear that the Lustre based Singularity and

Charliecloud are comparable, and sometimes better, when it comes to performance, in comparison to the local storage based Docker and Podman. This applies for both single-node as well as multi-node scenarios. In the multi-node scenario, the Lustre based tests showed comparable I/O performance to that of local storage tests.

In exploring our second research question of *how would the Lustre file system performance differ for different container solutions*, we are able to observe a large difference in the number of file and metadata requests that occur with Singularity and Charliecloud compared to Docker and Podman. This difference gets more pronounced with increasing number of processes, especially for Charliecloud. It is clear that the way Charliecloud is built (using a whole uncompressed Linux file tree) causes it to create a large number of MDS and OSS requests stemming from the large number of individual file requests for the tree format. This overhead increases with increasing levels of parallelism because each additional process needs to individually access the container related files. Singularity seems to be the better option for running containers on HPC file systems because of its single file nature, which greatly reduces the MDS and OSS requests needed for just the container functionality as parallelism increases. This is an important finding, and should be considered if the users expect to have an HPC environment with a large number of parallel jobs running from containers atop Lustre; using the wrong container solution could potentially lead to resource contention and bottlenecks on the MDS and OSS servers. The Charliecloud documentation does make note of the fact that there could be high metadata load when running the uncompressed images from a shared file system. However, we are the first to explicitly measure the extent of the problem and how it aggravates with increasing parallelism.

V. Related Work

Many works have performed analyses of container solutions focusing on resource usage and performance. Kovacs et al. [52] offer basic comparisons on a smaller scale between different container solutions. On a larger scale, Rudyy et al. [7] compare Docker, Shifter, and Singularity in an HPC environment, focusing on scalability and portability between different architectures. Younge et al. [21] specifically focus on comparing Singularity and Docker in a Cray system environment. Torrez et al. [53] talk about the performance similarities of Singularity, Charliecloud, and Shifter at the CPU, memory, and application level in HPC. Wharf [54] explores storing Docker container images in a distributed file system such that different layers can be shared among users. In contrast to these works, this paper focuses on the effect of file I/O operations from different container images stored on Lustre and local storage.

Arango et al. [22] compare CPU, memory, GPU, and disk I/O performance for LXC, Docker, and Lustre. In comparison, our work provides a much closer look at the details of a distributed file system's behavior under container operations. Le et al. [23] conduct a performance comparison for scientific simulation benchmarks between Singularity and bare-metal runs on the Comet supercomputer at SDSC [55]. Beltre et al. [56] and Saha et al. [57] evaluate the use of containerization in cloud infrastructure for HPC and the effects of different interconnects. These works are complementary to ours, wherein our work aims to fill the gap in knowledge about the behavior of different container solutions I/O on a parallel file system such as Lustre.

Huang et al. [25] perform a comparison of Lustre and GlusterFS as backing stores for a cloud system and conclude that Lustre is superior in performance and throughput. That is another motivation for using Lustre as the HPC file system for our analysis. Zhao et al. [24] offer a comparison of representative file systems for scientific applications and make the case for distributed metadata management. Pan et al. [26] are able to provide a framework for integrating PFS into cloud settings for use with HPC applications hosted in the cloud. All of these works evaluate file systems for the cloud setting, but do not cover the effect of containers on Lustre, which is analyzed in this paper.

VI. Conclusion

We have presented an empirical analysis of container solutions for HPC environments. We study four container solutions: *Docker, Podman, Singularity, and Charliecloud* on the widely popular Lustre file system. We present a framework for managing the analysis of the different container solutions, incorporating multiple benchmarks, and integrating the metrics collection from the clients as well as the multiple Lustre server nodes. Our evaluation shows startup time overhead for Docker and Podman, as well as network overhead at startup time for Singularity and Charliecloud. Our I/O evaluations show that with increasing parallelism, Charliecloud incurs large overhead on Lustre's MDS and OSS. Moreover, we find that the observed throughput of containers on Lustre is at par with containers running from local storage. This is promising. In future work, we plan to extend our analysis to HPC object stores, such as Ceph, and other parallel file systems, such as BeeGFS.

Acknowledgment

This work is sponsored in part by the National Science Foundation under grants CCF-1919113, CNS-1405697, CNS- 1615411, CNS-1565314/1838271.

References

[1] "14 Tech Companies Embracing Container Technology." https://learn.g2.com/container-technology. Accessed: November 30 2019.

[2] E. Gawehn, J. A. Hiss, and G. Schneider, "Deep Learning in Drug Discovery," *Molecular Informatics*, vol. 35, no. 1, 2016.

[3] M. Reichstein, G. Camps-Valls, B. Stevens, M. Jung, J. Denzler, N. Carvalhais, and Prabhat, "Deep learning and process understanding for data-driven Earth system science," *Nature*, vol. 566, no. 7743, pp. 195–204, 2019.

[4] G. González and C. L. Evans, "Biomedical Image Processing with Containers and Deep Learning: An Automated Analysis Pipeline," *BioEssays*, vol. 41, no. 6, p. 1900004, 2019.

[5] R. Chard, Z. Li, K. Chard, L. Ward, Y. Babuji, A. Woodard, S. Tuecke, B. Blaiszik, M. J. Franklin, and I. Foster, "DLHub: Model and Data Serving for Science," in *Proceedings of 2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, May 2019.

[6] P. Xu, S. Shi, and X. Chu, "Performance Evaluation of Deep Learning Tools in Docker Containers," in *Proceedings of the 3rd International Conference on Big Data Computing and Communications (BIGCOM)*, Aug. 2017.

[7] O. Rudyy, M. Garcia-Gasulla, F. Mantovani, A. Santiago, R. Sirvent, and M. Vázquez, "Containers in HPC: A Scalability and Portability Study in Production Biological Simulations," in *Proceedings of 2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, May 2019.

[8] W. Shin, C. D. Brumgard, B. Xie, S. S. Vazhkudai, D. Ghoshal, S. Oral, and L. Ramakrishnan, "Data Jockey: Automatic Data Management for HPC Multi-tiered Storage Systems," in *Proceedings of 2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, May 2019.

[9] C. Seamons, "Building complex software inside containers (poster)," in *2019 International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, ACM, Nov. 2019.

[10] G. M. Kurtzer, V. Sochat, and M. W. Bauer, "Singularity: Scientific containers for mobility of compute," *PloS one*, vol. 12, no. 5, p. e0177459, 2017.

[11] "Podman." https://podman.io/. Accessed: November 30 2019.

[12] L. Gerhardt, W. Bhimji, M. Fasel, J. Porter, M. Mustafa, D. Jacobsen, V. Tsulaia, and S. Canon, "Shifter: Containers for hpc," in *J. Phys. Conf. Ser.*, vol. 898, 2017.

[13] R. Priedhorsky and T. Randles, "Charliecloud: unprivileged containers for user-defined software stacks in HPC," in *Proceedings of the 2017 International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, (Denver, Colorado), ACM, Nov. 2017.

[14] A. K. Paul, O. Faaland, A. Moody, E. Gonsiorowski, K. Mohror, and A. R. Butt, "Understanding hpc application i/o behavior using system level statistics," 2019.

[15] B. Wadhwa, A. K. Paul, S. Neuwirth, F. Wang, S. Oral, A. R. Butt, J. Bernard, and K. Cameron, "iez: Resource contention aware load balancing for large-scale parallel file systems," in *Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2019.

[16] A. K. Paul, R. Chard, K. Chard, S. Tuecke, A. R. Butt, and I. Foster, "Fsmonitor: Scalable file system monitoring for arbitrary storage systems," in *2019 IEEE International Conference on Cluster Computing (CLUSTER)*, pp. 1–11, IEEE, 2019.

[17] A. K. Paul, B. Wang, N. Rutman, C. Spitz, and A. R. Butt, "Efficient metadata indexing for hpc storage systems.," in *CCGRID*, p. 10, 2020.

[18] M. G. Xavier, M. V. Neves, F. D. Rossi, T. C. Ferreto, T. Lange, and C. A. De Rose, "Performance evaluation of container-based virtualization for high performance computing environments," in *Proceedings of the 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing.*, IEEE, 2013.

[19] C. Ruiz, E. Jeanvoine, and L. Nussbaum, "Performance Evaluation of Containers for HPC," in *Proceedings of the 2015 European Conference on Parallel Processing*, 2015.

[20] J. Sparks, "Enabling Docker for HPC," *Concurrency and Computation: Practice and Experience*, Dec. 2018.

[21] A. J. Younge, K. Pedretti, R. E. Grant, and R. Brightwell, "A tale of two systems: Using containers to deploy hpc applications on supercomputers and clouds," in *Proceedings of the 2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, Dec. 2017.

[22] C. Arango, R. Dernat, and J. Sanabria, "Performance Evaluation of Container-based Virtualization for High Performance Computing Environments," *arXiv:1709.10140 [cs]*, Sept. 2017.

[23] "Performance Analysis of Applications using Singularity Container on SDSC Comet," in *Proceedings of the Practice and Experience in Advanced Research Computing 2017 on Sustainability, Success and Impact (PEARC).*, (New Orleans, LA, USA).

[24] D. Zhao, X. Yang, I. Sadooghi, G. Garzoglio, S. Timm, and I. Raicu, "High-Performance Storage Support for Scientific Applications on the Cloud," in *Proceedings of the 6th Workshop on Scientific Cloud Computing*, (Portland, Oregon, USA), June 2015.

[25] W.-C. Huang, C.-C. Lai, C.-A. Lin, and C.-M. Liu, "File System Allocation in Cloud Storage Services with GlusterFS and Lustre," in *Proceedings of 2015 IEEE International Conference on Smart City/SocialCom/SustainCom (SmartCity)*, Dec. 2015.

[26] "Integrating High Performance File Systems in a Cloud Computing Environment," in *2012 SC Companion: High Performance Computing, Networking Storage and Analysis*, (Salt Lake City, UT).

[27] A. K. Paul, A. Goyal, F. Wang, S. Oral, A. R. Butt, M. J. Brim, and S. B. Srinivasa, "I/O load balancing for big data HPC applications," in *International Conference on Big Data*, pp. 233–242, IEEE, 2017.

[28] A. K. Paul, S. Tuecke, R. Chard, A. R. Butt, K. Chard, and I. Foster, "Toward scalable monitoring on large-scale storage for software defined cyberinfrastructure," in *Proceedings of the 2nd Joint International Workshop on Parallel Data Storage & Data Intensive Scalable Computing Systems*, pp. 49–54, 2017.

[29] "Sysbench." https://github.com/akopytov/sysbench. Accessed: November 30 2019.

[30] "Coral benchmarks: Haccio." https://asc.llnl.gov/CORAL-benchmarks/#hacc. Accessed: November 30 2019.

[31] "LLNL - IOR Benchmark." https://asc.llnl.gov/sequoia/benchmarks/IORsummaryv1.0.pdf. Accessed: March 11 2019.

[32] "OpenSFS and EOFS - Lustre file system." http://lustre.org/. Accessed: March 23 2019.

[33] "Parallel Virtual File Systems on Microsoft Azure Part 2." https://bit.ly/2OGat4k. Accessed: November 30 2019.

[34] "Top 500 List." https://www.top500.org/lists/2019/11/. Accessed: November 30 2019.

[35] "Frontier." https://www.olcf.ornl.gov/frontier/#4. Accessed: 2020-03-03.

[36] "Kernel virtual machine." https://www.linux-kvm.org/page/Main_Page. Accessed: November 30 2019.

[37] "cgroups - linux man pages." http://man7.org/linux/man-pages/man7/cgroups.7.html. Accessed: November 30 2019.

[38] "namespaces - linux man pages." http://man7.org/linux/man-pages/man7/namespaces.7.html. Accessed: November 30 2019.

[39] "Docker." https://www.docker.com/. Accessed: November 19 2019.

[40] "LXC." https://linuxcontainers.org/lxc/. Accessed: November 25 2019.

[41] "Use the OverlayFS storage driver." https://docs.docker.com/storage/storagedriver/overlayfs-driver/. Accessed: November 25 2019.

[42] "Lustre graph driver for docker." https://github.com/bacaldwell/lustre-graph-driver.

[43] "What is Podman?." https://podman.io/whatis.html. Accessed: November 25 2019.

[44] "conmon." https://github.com/containers/conmon. Accessed: November 25 2019.

[45] "User namespaces support in Podman." https://www.projectatomic.io/blog/2018/05/podman-userns/. Accessed: November 25 2019.

[46] "SquashFS." https://github.com/plougher/squashfs-tools. Accessed: November 25 2019.

[47] "Buildah." https://buildah.io/. Accessed: May 19 2020.

[48] A. S. et al., "umoci - standalone tool for manipulating container images," 2016.

[49] "pivot_root(2): change root file system - Linux man page." Accessed. March 22 2020.

[50] "sar(1) - Linux man page." https://linux.die.net/man/1/sar.

[51] S. Habib, "Cosmology and Computers: HACCing the Universe," in *Proceedings of 2015 International Conference on Parallel Architecture and Compilation (PACT)*, Oct. 2015.

[52] Kovács, "Comparison of different Linux containers," in *Proceedings of the 40th International Conference on Telecommunications and Signal Processing (TSP)*, July 2017.

[53] A. Torrez, T. Randles, and R. Priedhorsky, "Hpc container runtimes have minimal or no performance impact," in *2019 IEEE/ACM International Workshop on Containers and New Orchestration Paradigms for Isolated Environments in HPC (CANOPIE-HPC)*, pp. 37–42, 2019.

[54] C. Zheng, L. Rupprecht, V. Tarasov, D. Thain, M. Mohamed, D. Skourtis, A. S. Warke, and D. Hildebrand, "Wharf: Sharing Docker Images in a Distributed File System," in *Proceedings of the 2018 ACM Symposium on Cloud Computing (SoCC)*, (CA, USA), pp. 174–185, Oct. 2018.

[55] "San Diego Supercomputer Center." https://www.sdsc.edu/. Accessed: November 30 2019.

[56] A. M. Beltre, P. Saha, M. Govindaraju, A. Younge, and R. E. Grant, "Enabling HPC Workloads on Cloud Infrastructure Using Kubernetes Container Orchestration Mechanisms," in *Proceedings of the 2019 IEEE/ACM International Workshop on Containers and New Orchestration Paradigms for Isolated Environments in HPC (CANOPIE-HPC)*, Nov. 2019.

[57] P. Saha, A. Beltre, P. Uminski, and M. Govindaraju, "Evaluation of Docker Containers for Scientific Workloads in the Cloud," in *Proceedings of the Practice and Experience on Advanced Research Computing (PEARC)*, July 2018.