On the Use of Containers in High Performance Computing Environments

Subil Abraham*[1], Arnab K. Paul*[1], 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 containerized solutions. As such data analysis is also being utilized in the high performance computing (HPC) domain, the need for container support in HPC has become paramount. However, container adoption in HPC face crucial performance and I/O challenges. One obstacle is that while there has been container solutions for HPC, such solutions have not been thoroughly investigated, especially from the aspect of their impact on the crucial I/O throughput needs of HPC. 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, especially how containers interact with the HPC storage systems. We present the design of an analytical framework that is deployed on all nodes in an HPC environment, and captures aspects such as 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 optimization opportunities, we can enhance performance of containers atop HPC and help the aforementioned applications.**

***Keywords*-Container, Charliecloud, Docker, High Performance Computing, Lustre File System, Podman, Singularity**

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, easily capture dependencies into easy-to-deploy images, provide application portability, and support massive scale. Thus, containers free the application developers from lower-level deployment and management issues, allowing them to focus on their applications and in turn significantly reduce the time-to-solution for mission-critical applications.

Modern data analysis is compute and I/O hungry [25], [35] requiring 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 are relying 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 environments. A challenge to this end is that such DL/ML software stacks are complex and bespoke, with no two setups exactly identical. This leads to a operations nightmare of addressing library, software, and other dependencies and conflicts. To this end, enterprise data analysis solutions are relying on containers, because containers can 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.

Containers are increasingly being adopted in the HPC environment, and have produced benefits for large scale image processing, DL/ML workloads [27], [24], [43], and simulations [36]. 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 [40] and Los Alamos [39] national labs. A number of container solutions have been realized for the HPC environment, such as Singularity [31], Podman [14], Shifter [26], and Charliecloud [34]. Among these, Singularity and Charliecloud make the best use of the 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. Moreover, different container frameworks introduce 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 will help build an effective ecosystem where both parallel file systems and containers can thrive. There have been several works that have looked into the performance of containers on HPC systems [42], [37], [36], [41], [44], [22], [32]. There have also been studies on parallel file systems for use in cloud environments [45], [29], [33]. However, none of the existing works have studied the behavior of HPC storage systems in service of containers.

In this paper, we examine the performance of different container solutions on a distributed file system. We present a framework to analyze different containerization solutions running atop a HPC storage system. The framework collects and compares performance metrics from client and storage

[1]Both authors have equal contribution.

nodes, including CPU, memory, network, and file system usage on individual nodes. We also collect metadata and I/O statistics gathered from the storage system. We focus on file I/O, specifically analyzing the effects of I/O operations made by containers on the storage system. We use the containerized image of Sysbench [17] to run CPU, memory, and file I/O benchmarks. We also use real-world representative HPC workloads such as HaccIO [5] and IOR [9].

We study four extant container solutions: Docker, Podman, Singularity, and Charliecloud. The HPC storage system selected for our analysis is Lustre, as it is one of the most widely used parallel file systems in HPC [13]. According to the latest Top 500 list [18], Lustre powers $\sim$ 60% of the top 100 supercomputers in the world, and will power Frontier [7], 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 storage 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 help both HPC practitioners and system administrators to obtain best I/O performance for running containerized solutions in HPC.

II. Background

In this section, we introduce and describe containers and some of the commonly used containerization solutions. We also describe HPC storage systems with an emphasis on the architecture of Lustre file system.

A. Containers

Containers differ from other virtualization methods like KVM [8] which explicitly virtualize the hardware and run separate kernels. The containers on the same node share the same kernel. It is able to provide a VM like isolation through the use of cgroups [2] and namespaces [12]. As a result, a container can provide the necessary isolation between applications without the heavy resource utilization of a virtual machine, allowing more containers to run simultaneously on a node.

Containers provide the ability to read and write (I/O) by *bind-mount*ing a directory in the container to a directory on the node where the container is being run. Therefore, reads and writes that occur on a directory inside the container will actually take place in the underlying directory it is bind mounted to on the underlying OS.

Beyond isolation, arguably the main benefit of containerization, particularly in a scientific, HPC environment is the ease of packaging the application and all of its dependencies in an easy to distribute format. The packaging even allows going all the way down to picking a suitable Linux distribution for the application. This is appealing to application developers as they do not have to worry about managing dependencies across a large HPC system, nor having to work within the confines of the existing module system.

B. Containerization Solutions

1) Docker: Docker [6] is one of the most popular containerization solutions. Although containerization solutions like LXC [11] existed before, it was Docker that popularized containerization as a better and more efficient solution for providing isolation for applications, especially in the cloud sphere. Figure 1a shows the simplified architecture of Docker.

There are however some drawbacks to Docker that hold it back from enthusiastic adoption in HPC environments. Firstly. It 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 very bloated. It also needs root permissions to be able to run. This can be harmful in HPC environments where arbitrary code can be executed. 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 [19] to create a union of all its layers and creates a read-write layer at the top. However, a distributed file system like Lustre does not support OverlayFS. So an image must be explicitly downloaded and run from a supported local file system. Docker also does not provide useful support for native MPI operations.

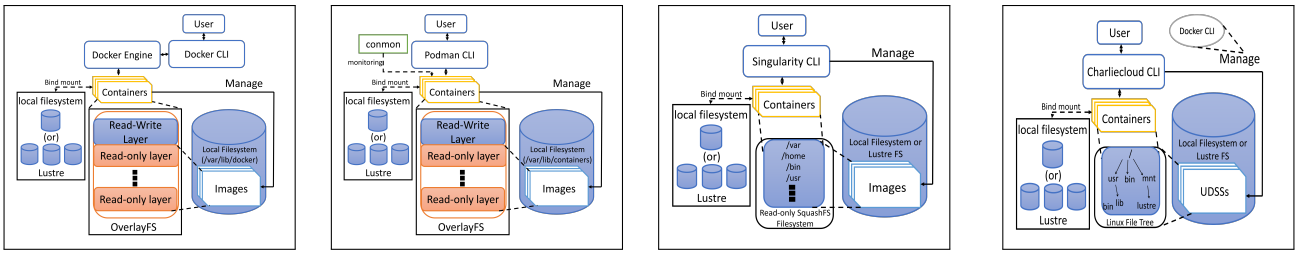
There have been solutions put forth to allow storing Docker images in the Lustre file system. These involve an inefficient way of abusing the device-mapper driver and creating loopback devices in Lustre so that the data-root can be pointed to those devices [10]. The device-mapper driver is now deprecated and will be removed in a future Docker releases, so the solution is not future-proof.

2) Podman: Podman [21] was developed out of the need for a container management tool to break away from the requirement of needing a Docker daemon with root privileges to manage containers and images. It provides a comprehensive container solution equivalent to the Docker ecosystem, without Docker's dependence on a single daemon.

Podman replaces the daemon-client architecture of Docker with individual processes running the containers while using *conmon* [4] to provide the necessary monitoring and debugging capabilities, as seen in Figure 1b.

In addition, Podman can run rootless containers through the use of user namespaces [20] which ensures additional security and separation among containers, especially in HPC scenarios.

Podman runs into the same problem as Docker where it needs OverlayFS to store and run containers. The layer structure of the images is OCI compliant but it serves as a roadblock for storing images on distributed file systems in HPC environments.



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

Fig. 1: Architecture of the container solutions.

3) Singularity: Singularity [31] is a container framework tailored specifically for HPC environments. Its goals are to provide isolation for workloads while preventing privilege escalation, native support for MPI, Infiniband, and GPUs, and ease of portability and reusability through distributing container images as a single squashfs file. This is also Singularity's biggest benefit because this allows for storing images and running containers from the Lustre file system. It just uses a single image file and does not have the layer structure where it would need OverlayFS which is incompatible with Lustre, as seen in Figure 1c, .

Singularity also has a far greater focus on the security features, such as root file system encryption, as well as cryptographically signing containers images. Singularity containers are by default read-only and all write operations can only be done via bind-mounted directories.

4) Charliecloud: Charliecloud [34] is a container framework from Los Alamos National Laboratory also intended primarily for use in HPC. Its goals are similar to Singularity and also provides native MPI support. It differs from Singularity in that it focuses on simplicity of architecture instead of portability. Its primary aim is to provide a way to encapsulate dependencies with minimal overhead. It depends on the presence of Docker to build an image and then extracts all the contents of that image into a file tree called a User Defined Software Stack (UDSS). As a result, instead of existing as compressed layers (as in Docker and Podman) or as a single file (as in Singularity), it exists as a file tree from which the application is run by *chroot*ing [3] into it, as shown in Figure 1d. 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 somewhat trusted environment.

Discussion: Although Docker and Podman were built for a more cloud-based environment and not necessarily built for HPC scenarios, it is still important to study them in HPC environments as they are well-known and widely used. Also, the container solutions built for HPC, support the use of Docker images and sometimes (like Charliecloud) explicitly require Docker as a dependency.

Another prominent container solution in the HPC field is Shifter [26]. We created and evaluated a mockup of Shifter, but it was only a coarse approximation and did not produce any useful comparable results due to Shifter's architectural closeness to NERSC's supercomputers and their job scheduling setup, which is difficult to recreate outside of their facilities. Therefore, the evaluation results for Shifter are not discussed in this paper.

C. HPC Storage Systems

HPC storage systems are designed to distribute file data across multiple servers so that multiple clients can access a file system simultaneously. Typically, it consists of *clients* that read or write data to the file system, *data servers* where data is stored, *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, for whom it seems as though they are accessing a local file system. Therefore, important functions of a distributed file system include avoiding potential conflict among multiple clients and ensuring data integrity and system redundancy. The most common HPC file systems include Lustre, GlusterFS, BeeGFS, and IBM Spectrum Scale, with Lustre being the most represented in HPC environments.

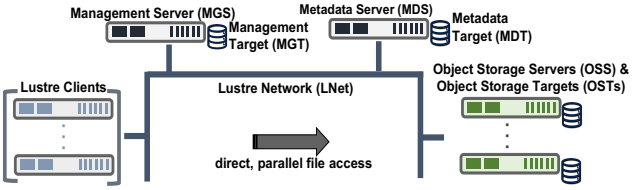


Fig. 2: An overview of Lustre architecture.

1) Lustre File System: The architecture of the Lustre file system is shown in Figure 2. 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 in a Lustre file system. 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* which 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. Analytical Framework

A. Deployment of Lustre File System

We use a Lustre file system cluster of 10 nodes with 1 MDS, 7 OSSs and 2 clients. 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 bandwidth ethernet. Furthermore, each OSS has 5 OSTs, each supporting 10 GB of attached storage. Therefore, our analysis is done on a 350 GB Lustre store.

B. Container Deployment

For container deployment on HPC, since Docker and Podman need OverlayFS which is not supported by Lustre, they use the client node's local storage to store container images. For Singularity and Charliecloud containers, the analysis was done with them running on the Lustre file system, except for startup time tests and the final multi-node throughput tests which were evaluated on both client node's local storage as well as the Lustre file system. The analysis included benchmarking CPU, memory, and file I/O performance of the containers with Sysbench [17], as well as looking at the Lustre file system behavior and I/O throughput under a real-world HPC workload through the lens of containers using HaccIO [5] and IOR [9].

C. Metrics Collection Framework

A framework was built to encapsulate all the required operations of building the container images, running them, collecting and analyzing the metrics. The framework is shown in Figure 3.

For the analysis, the framework takes options for which containerization 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 analysis framework incorporates the collection of `sar` [16] 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 analytical framework is activated, the framework spins up servers on the Lustre nodes that will listen for GET

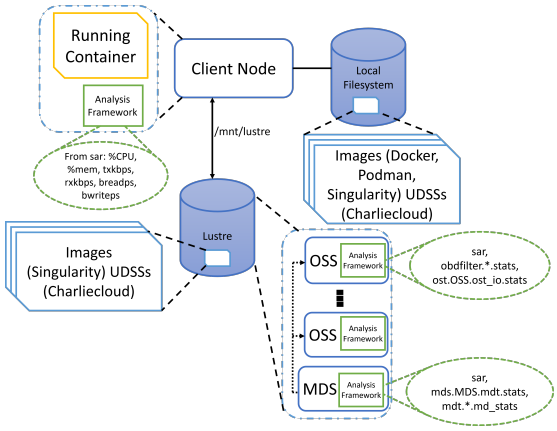


Fig. 3: Analysis framework.

requests from the client. The GET request is sent from the client node at the start and stop of each benchmark run to trigger the start and stop of metrics collection on the Lustre nodes. 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. All Sysbench benchmarks are cut off at 30 seconds and are forcibly stopped if they overrun. The HaccIO and IOR benchmark is 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

The setup involved a Lustre file system deployment of 7 OSSs, 1 MDS and container operations were done through two client nodes to validate both single and multi-node setups. Our single-node workloads include testing memory operations, CPU utilization, and sequential and random file I/O through the use of Sysbench.

The CPU utilization workload from Sysbench repeatedly calculates prime numbers up to 1,000,000 to stress load the CPU until the specified timeout is reached.

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 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 it, depending on the particular file I/O benchmark being run. All Sysbench benchmarks are run with 8 threads and are averaged across 10 runs.

In addition, we also tested a real-world multi-node setup using HaccIO [5] and IOR [9] running simultaneously from two different client nodes. The HaccIO benchmark simulates the I/O patterns of the Hardware Assisted Cosmology Code [28],

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.

B. Research Questions

In analyzing the containers, we try to answer the following research questions:

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

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

C. Container Startup Time

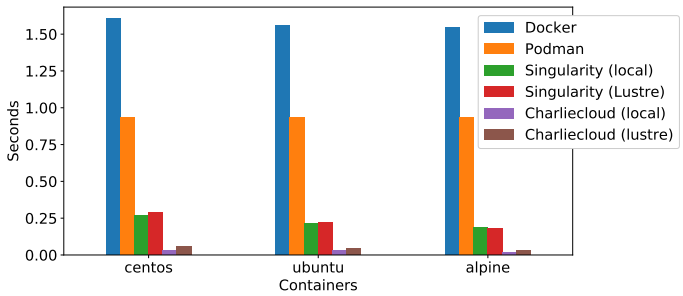


Fig. 4: Startup times for OS images for different container solutions.

Figure 4 shows the time to start up for different representative container images. 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.

D. Observations on The Client

1) 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 has additional CPU overhead at the startup time. This also explains the longer startup time for Docker and Podman compared to the others.

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 the Lustre file system. The single file nature causes Singularity to load a bulk of the data from the image at startup time causing the large initial data transfer.

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 it will need to access a lot of individual files from the image tree on the Lustre file system right at the outset.

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

2) CPU and Memory Utilization: Figure 6 and 7 show box plots of the CPU (sbcpu), memory (sbmem) and I/O (sb-fileio) workloads, and their corresponding CPU and memory utilization, across 10 runs. All four container solutions are fairly similar in how they use CPU and memory for different workloads.

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

E. Behavior of Lustre File System

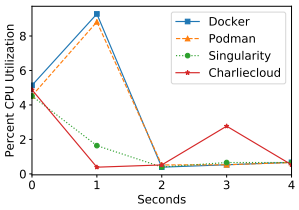
Table II shows the Lustre statistics for the Sysbench Random Read-Write file I/O benchmark.

Singularity has the highest amount of data read per OSS. This is mainly due to the fact that Singularity is reading larger blocks of the contents from its container image in the Lustre file system, in addition to the read operation of the benchmark itself. Docker and Podman reads container images from the client's local file system and so they don't have that overhead. Singularity reads larger blocks compared to Charliecloud since it is a large single file and would load more data at a time, whereas Charliecloud reads smaller amounts of data since it just has to read individual files from its UDSS file tree.

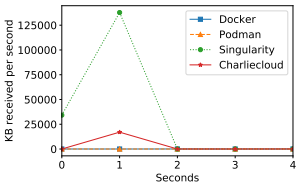
For the same reasons, Singularity also shows a higher average number of requests on the MDS and OSSs. However, this is in contrast to the multinode real world scenario where Charliecloud has comparatively higher number of OSS and MDS requests, as discussed in Section IV-F. In either case, the higher 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 Benchmark - HaccIO and IOR

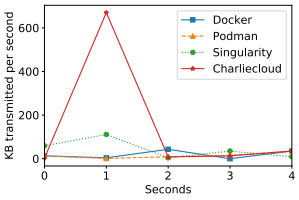
As a final step, we also obtain the I/O metrics across 10 HaccIO and IOR runs on the Lustre file system run through each container solution in a multinode setup, with HaccIO and IOR running separately on each client node to mimic a heterogeneous workload on Lustre. The number of MPI processes are varied to see how the throughput changes with increased parallelism in the workloads. In the workload, both



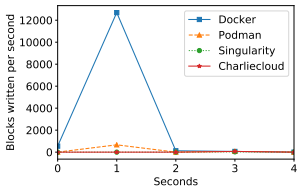
(a) Spike in CPU utilization for Docker and Podman in sysbench sequential readwrite.



(b) Spike in data received over network for Singularity in sysbench memory random read[1].



(c) Spike in data transmitted over network for Charliecloud in sysbench memory random read[1].



(d) Spike in blocks written for Docker in sysbench sequential readwrite.

Fig. 5: Spikes observed in different metrics during container startup for different 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.

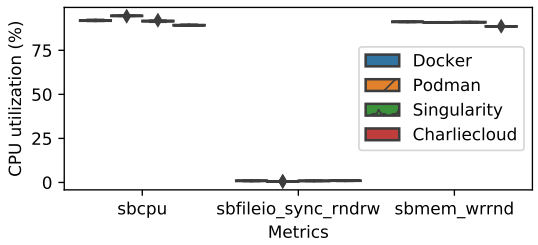


Fig. 6: CPU utilization for different workloads.

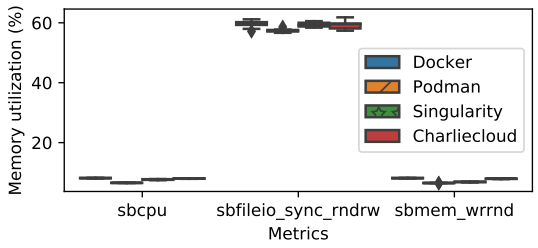


Fig. 7: Memory utilization for different workloads.

HaccIO and IOR are run simultaneously in the multinode setup with 2, 4, and 8 MPI processes each.

Figures 8 and 9 show that Charliecloud, with increasing numbers of processes, has the highest amount of activity on OSSs and MDS, significantly higher than any other container framework, due to its UDSS structure. UDSS is a full file tree stored in Lustre which 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. Singularity on the other hand is stored as just one file and therefore, does not incur the overhead

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).

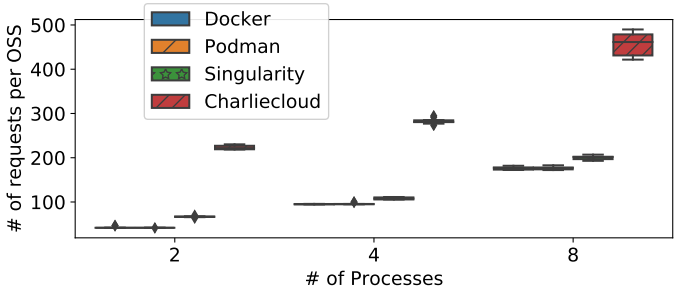


Fig. 8: Box plot of number of requests per OSS across 10 runs, averaged across 7 OSSs, for HaccIO and IOR benchmark with different number of MPI processes

of many individual file reads as in Charliecloud. Therefore, Singularity scales better on increasing the parallelism. For a single process, Charliecloud fares better as it makes fewer requests, as evidenced from Table II and is more efficient than Singularity, but this is not the case in a multi-process setup.

We performed another evaluation of HaccIO and IOR in the multinode 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 1 to 4. This means

[1]memory read benchmark itself doesn't do any file or network I/O.

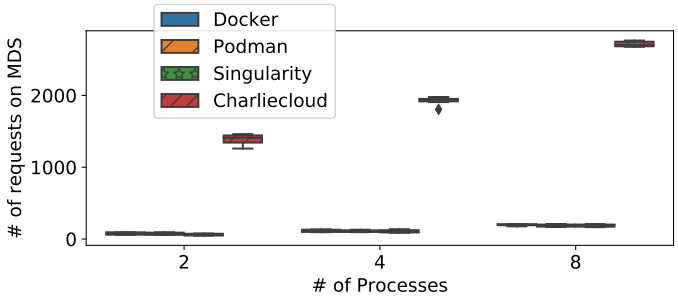


Fig. 9: Box plot of number of requests across 10 runs in MDS for HaccIO and IOR benchmark with different number of MPI processes

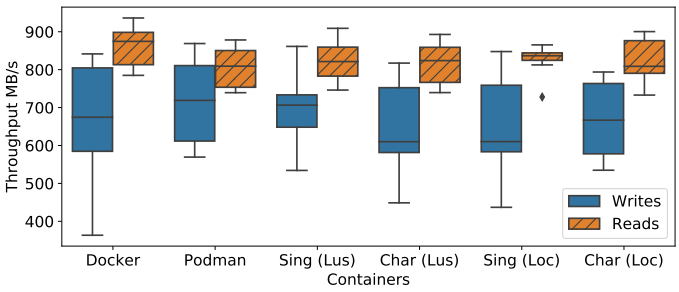


Fig. 10: Box plot showing write and write throughput for IOR during multinode run (Sing - Singularity, Char - Charliecloud, Lus - image stored on Lustre, Loc - image stored on client's local storage)

that each run has 8 MPI processes and the total IOR data written and read is 8 GB.

This setup allows a better look at the I/O throughput. Additionally, we are performing a run of Singularity and Charliecloud from the local storage, similar to the Docker and Podman set up, to let us to see if there is difference in throughput between running the same container from local storage and shared storage. 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 and local file system. This indicates that running these container images from a distributed file system is in no way detrimental to their I/O performance.

G. Discussion

To answer our first research question on *how would performance among different container solutions differ*, the most important observation is the significant difference in start-up times among all three container solutions as well as the spikes seen in the various metrics during container startup. Charliecloud is the least affected and therefore performs the best during container start-up. Besides start-up, 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 experiments show comparable I/O performance to experiments running from local storage, even though they have to deal with the additional network and distributed file system overhead.

In exploring our second research question on *how would the Lustre file system's 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 (a whole Linux file tree called a UDSS) causes Lustre to be heavily hit in the number of MDS and OSS requests because of the large number of individual file opens that Charliecloud has to deal with. 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, greatly reducing the MDS and OSS requests needed for just the container functionality as parallelism increases. This is an important finding to be mindful of especially if you expect to have an HPC environment with a large number of parallel jobs running from containers on Lustre as it could potentially lead to resource contention and bottlenecks on the MDS and OSS servers.

V. Related Work

Many works have performed analyses of container solutions focusing on resource usage and performance. Kovacs et al. [30] perform basic comparisons on a smaller scale between different container solutions. On a larger scale, Rudyy et al. [36] perform a comparison of Docker, Shifter, and Singularity in an HPC environment, focusing on scalability and portability between different architectures. Younge et al. [44] specifically focus on comparing Singularity and Docker in a Cray system environment. Wharf [46] explores storing Docker container images in a distributed file system such that different layers can be shared among users. Our work instead focuses on the effect of file I/O operations from different container images stored on Lustre and local storage. Arango et al. [22] perform a comparison of 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. [32] conduct a performance comparison for scientific simulation benchmarks between Singularity and bare-metal runs on the Comet supercomputer at SDSC [15]. Beltre et al. [23] and Saha et al. [38] evaluate the use containerization solutions in cloud infrastructure for HPC and the effects of different interconnects. None of the works focus the analysis of container I/O on a distributed file system like Lustre, which is the gap that our work aims to fill.

Huang et al. [29] 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 storage system for our analysis. Zhao et al. [45] perform a comparison of representative file systems for scientific applications and make the case for distributed metadata management. Pan et al. [33] are able to provide a framework for integrating parallel file systems into the cloud for use with high-performance applications. All of these works that evaluate file systems for cloud do not cover the effect of containers, a now widespread tool used in clouds, on Lustre. Our work aims to fill this gap between containers and HPC storage systems.

VI. Conclusion

This work aims to fill the gap in the literature between containers and HPC storage systems by performing an efficient empirical analysis of container behavior for four container solutions: *Docker, Podman, Singularity, and Charliecloud* on the widely popular Lustre file system. We present an analytical 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 evaluations show some startup time overhead for Docker and Podman, as well as network overhead at startup time for Singularity and Charliecloud. Our I/O evaluations shows Charliecloud has heavy container overhead of MDS and OSS requests when running containers from images stored on Lustre, with increase in parallelism. However, the observed throughput of containers on Lustre is still on par with containers running from local storage. In future, we plan to extend our analysis to HPC object stores, like Ceph and other parallel file systems, like BeeGFS.

VII. Acknowledgements

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] cgroups - linux man pages. http://man7.org/linux/man-pages/man7/cgroups.7.html. Accessed: November 30 2019.
[3] chroot(1) - Linux man page. https://linux.die.net/man/1/chroot.
[4] conmon. https://github.com/containers/conmon. Accessed: November 25 2019.
[5] Coral benchmarks: Haccio. https://asc.llnl.gov/CORAL-benchmarks/#hacc. Accessed: November 30 2019.
[6] Docker. https://www.docker.com/. Accessed: November 19 2019.
[7] Frontier. https://www.olcf.ornl.gov/frontier/#4. Accessed: 2020-03-03.
[8] Kernel virtual machine. https://www.linux-kvm.org/page/Main_Page. Accessed: November 30 2019.
[9] LLNL - IOR Benchmark. https://asc.llnl.gov/sequoia/benchmarks/IORsummaryv1.0.pdf. Accessed: March 11 2019.
[10] Lustre graph driver for docker. https://github.com/bacaldwell/lustre-graph-driver.
[11] LXC. https://linuxcontainers.org/lxc/. Accessed: November 25 2019.
[12] namespaces - linux man pages. http://man7.org/linux/man-pages/man7/namespaces.7.html. Accessed: November 30 2019.
[13] Parallel Virtual File Systems on Microsoft Azure Part 2. https://bit.ly/2OGat4k. Accessed: November 30 2019.
[14] Podman. https://podman.io/. Accessed: November 30 2019.
[15] San diego supercomputer center. https://www.sdsc.edu/. Accessed: November 30 2019.
[16] sar(1) - Linux man page. https://linux.die.net/man/1/sar.
[17] Sysbench. https://github.com/akopytov/sysbench. Accessed: November 30 2019.
[18] Top 500 List. https://www.top500.org/lists/2019/11/. Accessed: November 30 2019.
[19] Use the OverlayFS storage driver. https://docs.docker.com/storage/storagedriver/overlayfs-driver/. Accessed: November 25 2019.
[20] User namespaces support in Podman. https://www.projectatomic.io/blog/2018/05/podman-userns/. Accessed: November 25 2019.
[21] What is Podman? https://podman.io/whatis.html. Accessed: November 25 2019.
[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. arXiv: 1709.10140.
[23] A. M. Beltre, P. Saha, M. Govindaraju, A. Younge, and R. E. Grant. Enabling HPC Workloads on Cloud Infrastructure Using Kubernetes Container Orchestration Mechanisms. IEEE/ACM CANOPIE-HPC'19, pages 11–20, Denver, CO, USA, Nov. 2019. IEEE.
[24] 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. IPDPS'19, pages 283–292, May 2019.
[25] E. Gawehn, J. A. Hiss, and G. Schneider. Deep Learning in Drug Discovery. *Molecular Informatics*, 35(1):3–14, 2016.
[26] 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.*, volume 898, page 082021, 2017.
[27] G. González and C. L. Evans. Biomedical Image Processing with Containers and Deep Learning: An Automated Analysis Pipeline. *BioEssays*, 41(6):1900004, 2019.
[28] S. Habib. Cosmology and Computers: HACCing the Universe. PACT'15, pages 406–406, Oct. 2015. ISSN: 1089-795X.
[29] W.-C. Huang, C.-C. Lai, C.-A. Lin, and C.-M. Liu. File System Allocation in Cloud Storage Services with GlusterFS and Lustre. IEEE SmartCity'15, pages 1167–1170, Dec. 2015.
[30] Kovács. Comparison of different Linux containers. TSP'17, pages 47–51, July 2017.
[31] G. M. Kurtzer, V. Sochat, and M. W. Bauer. Singularity: Scientific containers for mobility of compute. *PloS one*, 12(5):e0177459, 2017.
[32] E. Le and D. Paz. Performance Analysis of Applications using Singularity Container on SDSC Comet. PEARC'17, pages 1–4, New Orleans, LA, USA, 2017.
[33] A. Pan, J. P. Walters, V. S. Pai, D.-I. D. Kang, and S. P. Crago. Integrating High Performance File Systems in a Cloud Computing Environment. SCC'12, pages 753–759, Salt Lake City, UT, Nov. 2012.
[34] R. Priedhorsky and T. Randles. Charliecloud: unprivileged containers for user-defined software stacks in HPC. SC '17, pages 1–10, Denver, Colorado, Nov. 2017. ACM.
[35] 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*, 566(7743):195–204, 2019.
[36] 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. IPDPS'19, pages 567–577, May 2019. ISSN: 1530-2075.
[37] C. Ruiz, E. Jeanvoine, and L. Nussbaum. Performance evaluation of containers for hpc. Euro-Par'15, pages 813–824. Springer, 2015.
[38] P. Saha, A. Beltre, P. Uminski, and M. Govindaraju. Evaluation of Docker Containers for Scientific Workloads in the Cloud. In *PEARC '18*, pages 1–8, Pittsburgh, PA, USA, July 2018.
[39] C. Seamons. Building complex software inside containers (poster). SC'19, page 36. ACM, 2019.
[40] 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. IPDPS'19, pages 511–522, May 2019. ISSN: 1530-2075.
[41] J. Sparks. Enabling Docker for HPC. *Concurrency and Computation: Practice and Experience*, Dec. 2018.
[42] 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. PDP'13, pages 233–240. IEEE, 2013.
[43] P. Xu, S. Shi, and X. Chu. Performance Evaluation of Deep Learning Tools in Docker Containers. BIGCOM'17, pages 395–403, Aug. 2017.

[44] 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. IEEE CloudCom'17, pages 74–81. IEEE, 2017.

[45] D. Zhao, X. Yang, I. Sadooghi, G. Garzoglio, S. Timm, and I. Raicu. High-Performance Storage Support for Scientific Applications on the Cloud. ScienceCloud'15, pages 33–36, Portland, Oregon, USA, 2015.

[46] 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. SoCC'18, pages 174–185, CA, USA, 2018.