Scaling Containerization on multi-Petaflops CPU and GPU HPC platforms

Amit Ruhela[0000-0001-6547-714X], Stephen Lien Harrell [0000-0002-7304-6056], Richard Todd Evans [0000-0003-4992-9614]
Texas Advanced Computing Center
Austin, Texas
E-mail : {aruhela, sharrell, rtevans}@tacc.utexas.edu

***Abstract*—Containerization technologies provide a mechanism to encapsulate applications and many of their dependencies, facilitating software portability and reproducibility on HPC systems. However, in order to access many of the architectural features that enable HPC system performance, compatibility between certain components of the container and host is required, resulting in a trade-off between portability and performance. In this work, we discuss our experiences running two state-of-the-art containerization technologies on four leading petascale systems. We present how we build the containers to ensure performance and security and their performance at scale. We ran microbenchmarks at a scale of 6,144 nodes containing 0.35M MPI processes and baseline the performance of container technologies. We establish the near-native performance and minimal memory overheads by the containerized environments using MILC - a lattice quantum chromodynamics code at 139,968 processes and using VPIC - a 3d electromagnetic relativistic Vector Particle-In-Cell code for modeling kinetic plasmas at 32,768 processes. We demonstrate an on-par performance trend at a large scale on Intel, AMD, and three NVIDIA architectures for both HPC applications.**

***Index Terms*—Petascale, HPC, Containerization, Cloud Computing, Singularity, Charliecloud**

I. Introduction

Containerization is a powerful tool for scientific software development and portability across systems. It considerably reduces the time to build, test, and deploy applications by encapsulating code and dependencies together, allowing them to run on diverse platforms with minimal additional efforts. High Performance Computing (HPC) infrastructures provide tremendous computing capabilities along with optimized message communication actualized through advanced features like eager communication, shared memory, and Remote Direct Memory Access making them ideal for intensive scientific computation but challenging for software portability. Containers provide a promising way to hide system-level complexities, allowing researchers to focus on productive studies that include COVID-19 research, climate modeling, agriculture, healthcare, smart cities, e-commerce, deep learning, etc.

Containerization is a light-weight, low-overhead alternative to full machine virtualization. With Docker's [1] introduction in 2013, containerization gained tremendous popularity. Since then, several containerization techniques have been developed primarily based on chroot, control groups, and Linux namespace features. Table I compares three state-of-the-art containerization approaches - Docker, Singularity and Charliecloud. Docker is a user-friendly industry-standard containerization approach designed to support stateful microservices. This stateful approach creates security concerns on HPC systems due to its need for root privileges. The security issues combined with a lack of Message Passing System (MPI) support and resulting scaling limitations make Docker unfit for an HPC environment. Singularity and Charliecloud take different approaches and are designed for HPC users. Once installed with root privileges, Singularity and Charliecloud users can run respective containers without elevated permissions.

Several studies in the past have focused on the performance characterization of containerized workloads [2]–[7]. These studies, conducted at small problem sizes, indicate near-native performance by container-based techniques. However, none of the prior studies have comprehensively shown the performance, usability, and portability of state-of-the-art container approaches at medium and large scale. This motivates us to study the following two questions: **(1) Does the performance of container-based solutions on HPC clusters match bare-metal runs at varying problem scales? (2) What are the challenges and possible directions to exploit the state-of-the-art container techniques at a massive scale?**

TABLE I
Features of Containers

Attribute	Namespaces	Cgroups	User Escalation	Default Network	Root daemon	Keep changes after restart	Suitable for HPC
Docker	✓	✓	✓	Bridge	✓	✗	✗
Singularity	✓	✓	✗	Host	✗	✓	✓
Charliecloud	✓	✗	✗	Host	✗	✓	✓

A. Contributions

To the best of our knowledge, this is the first study investigating the performance of containers at HPC petascale. The main contributions of this paper are:

1) We present the challenges and possible approaches to build HPC clouds with container-based approaches.
2) We present the changes required to adapt containerization approaches to HPC infrastructures.

3) We establish the usability and portability of two user-defined containerization stacks (Singularity, Charliecloud) at various problem scales.
4) We compared the performance of state-of-the-art containers at a scale of 6,144 HPC nodes containing 344,064 processes with MPI microbenchmarks.
5) We compared the performance of native and container environments with two HPC scientific applications at up to 138,968 processes on 2,592 nodes.
6) We establish the performance of two state-of-the-art containers on four diverse HPC architectures: NVIDIA Quadro RTX 5000, V100, Intel Cascade Lake, and AMD Rome).

The rest of the paper is organized as follows: Section II presents the prior research works and establishes the novelty and basis of research conducted in this paper. Section III presents the background of the Singularity and Charliecloud container technologies and describes the benchmarks and applications experimented in this paper. Section IV-A presents the experimental setups and software configurations. Section IV-B and Section IV-C provide detailed experimental evaluations with microbenchmarks and HPC scientific applications. Finally, the conclusion are presented in Section V.

II. Related Work

The technology landscape of containerization started with the chroot system call in 1979 and was followed by FreeBSD Jails in 2000, the Linux VServer in 2001, Solaris Containers in 2004, Open VZ in 2005, Process Containers in 2006, Linux Containers(LXC) in 2008, Warden in 2011, and Google's Let Me Contain That For You (LMCTFY) in 2013. Containerization then became enormously popular with Docker's introduction in 2013. Since then, tremendous efforts have been made by researchers and industry to develop performant, secure, and portable container techniques for both Cloud and HPC environments.

In an early research paper by Xavier et.al. [5] from 2013, the trade-offs between performance and isolation in Linux VServer, OpenVZ, and LXC containers compared with traditional hypervisor-based Xen virtualization are presented. Later, Carlos et al. [4] in 2017 evaluated LXC, Docker, and Singularity's performance through a customized single node HPL-Benchmark and an MPI-based application on a multi-node testbed. They also studied application-level performance using a NAMD benchmark on a single GPU device attached with an eight-core processor. In the same year, Younge et al. [2] compared Singularity's performance on a Cray XC-series supercomputer and Docker on Amazon's Elastic Compute Cloud (EC2) and reported significant overheads in the cloud environment mainly due to the use of Ethernet rather than the Cray Aries interconnect.

In more recent studies, Hu et al. [8] investigated CPU, memory, and network bandwidth of Singularity containers whereas Rudyy et al. [9] explored the scalability and portability aspects of Docker, Singularity, and Shifter in the biological systems using Alya code at 256 nodes. At the benchmarks level on a medium system scale, Torrez et al. [10] demonstrated minimal overheads by Charliecloud, Shifter, and Singularity containers. In another interesting work, Cérin et.al. in Ref. [11] proposed a pervasive methodology for containerization of HPC jobs schedulers that shows better management of system resources in an economical way.

Apart from performance studies, Canon et al. in [12] reviewed the challenges and gaps in existing containerized approaches for HPC applications. A survey by Bachiega et al. [13] on recent research and challenges revealed a lack of thorough studies involving containers and their performance in the HPC environment. In light of this research and need, our focus in this paper is to bridge the gap in prior research work to establish the performance, portability, and usability of containers in the HPC environment using both microbenchmarks as well as HPC applications with real workloads. We present rigorous and comprehensive performance evaluations at petascale on four leading Intel, AMD and NVIDIA, architectures.

This paper is an extension of our earlier findings published in [14], [15]. We have run extensive experiments with scientific applications at much larger and diverse system scales.

III. Methodology

Out of several container runtimes, we evaluate Charliecloud and Singularity in this work. The goal is not to investigate comprehensively but to manifest the simplicity, usefulness, and performance of a few popular container types at petascale clusters.

Containerization on HPC clusters is challenging mainly due to access privileges and security requirements. Further, batch processing of jobs along with container overheads adds unique challenges to their usability. Portability of containers is restricted by Abstract Binary Interface (ABI) compatibility between the container and host hardware driver libraries along with instruction compatibility with host architecture (high speed interconnect drivers, GPU drivers, processor ISAs, processor specific compiler optimizations). For actable non-optimal performance, the container need not utilize specialized drivers and hardware capabilities and only ISA portability is required.

Singularity is a container platform specifically crafted for HPC systems. Similar to other user space container systems, Singularity bind mounts a container image and changes the apparent root (chroot) to the container. Singularity goes a step further to support the HPC ecosystem by mounting native devices (e.g., GPU, network, IB) and configured filesystem paths while also preserving Linux namespaces and user mapping inside the container. Singularity does not run a daemon service, but must be installed by the root user for privilege escalation. After building images from their own development systems, or on HPC if fakeroot is configured, users can pull images built with Singularity or Docker, and safely run them on shared HPC resources. While images can be stored in the cloud, they exist as single files on a filesystem, allowing them to be shared and managed like all other files.

Charliecloud, a user defined software stack (UDSS), exploits user and mount namespaces of Linux to run containers without needing privileged operations and/or daemons. Any packaging software capable of producing a standard Linux filesystem can build container images that can be hosted on private or public repositories (Dockerhub, Gitlab, NVIDIA NGC, etc.). Charliecloud is a 800 lines of open source code that demands minimal system control (sysctl) commands [16] to configure on computing facilities, which elude most security risks.

A. Microbenchmarks and Applications

We evaluate the performance of all the container technologies at the micro-benchmark level with Intel MPI Benchmarks (IMB) and at the application-level with two well-known HPC scientific applications - MILC and VPIC.

IMB [17] is a suite of MPI benchmarks that perform performance measurements for point-to-point and global communication operations for a range of message sizes. We use the standard MPI_Bcast Latency benchmark, which measures the one-way latency of the MPI broadcast operation. All experiments are performed at least three times with one Processes per node (PPN) and a full subscription for 10 to 1000 repetitions at various message sizes.

MIMD Lattice Computation (MILC) [18] is a Quantum Chromodynamics (QCD) code that is used in the study of strong interactions of subatomic physics to understand atomic nuclei, the evolution of the early universe, and connections with condensed matter physics. QCD describes the interaction of fundamental matter particles called quarks and force carriers called gluons, which bind to form the composite, hadronic particles, such as protons and neutrons. Lattice QCD (LQCD) is a numerical approach to QCD that approximates space and time by a 4D lattice. Physical quantities are computed by evaluating high-dimensional integrals using Molecular Dynamics and Markov-chain Monte Carlo methods. It's an open-source C89 code that utilizes the Highly-Improved Staggered-Quark (HISQ) formulation of LQCD. All the core components can be offloaded to GPUs through the QUDA library. On CPU architecture (Cluster C), MILC is scaled to run with 72x72x72x144 lattice at 17K, 35K, 70K, and 140K processes. On GPU architecture (Cluster A), MILC is run with 36x36x64x64 lattice on 32, 64, 128, and 256 V100 devices. Performance numbers are reported for time to solve Conjugate gradient, entire computations (Total Time), Linux reported time in seconds, and memory consumption.

The Vector Particle-in-Cell (VPIC) model is a particle-in-cell, first principles plasma physics application. It uses a structured grid and compute particles and electromagnetic fields [19]. An unreleased VPIC 2.0 beta is used, which has been ported to the Kokkos [20] performance portability framework [21]. The Kokkos OpenMP and CUDA backends are used to perform the benchmarking runs. The dataset used is 2D and uses all features of VPIC. The GPU experiments use 31.1 million particles, and 88 million particles are used for the CPU experiments. Performance is reported as overall runtime in seconds.

IV. Performance Evaluation

This section describes the experimental setup, provides the results of our experiments and presents an in-depth analysis of performance results. Running experiments on four different clusters ensures the generality of our performance analysis.

We use four distinct approaches to compare containerization overheads. In Section IV-B2, we start with baselining overheads at the MPI initialization level and then investigate the overheads at the collective communication level using MPI broadcast operation. We then compare the overheads at the container technology level with MPI Alltoall collective operation. Following in Section IV-C we investigate the overheads at the application level with diverse hardware architectures on petascale systems.

A. Experimental Setup

Cluster Configurations :

Cluster A : IBM OpenPOWER + InfiniBand (IB) + V100 : Each node on system contains dual socket Power-9 processors with 20 physical cores on each socket operating at 2.4 GHz, and contains 256GB DDR4 and 900GB of local temporary storage. The interconnect is Mellanox EDR (100Gb/s) InfiniBand with OFED version 4.5-2.2.9.0. The operating system is RHEL v7.6 with kernel version Linux 4.14.0-115.10.1.el7a.ppc64le. Each node contains four NVIDIA V100 GPUs, each having 16GB GDDR6 memory.

Cluster B : AMD Rome + InfiniBand : Each node on the Purdue Bell system contains dual socket AMD Rome processors with 64 physical cores on each socket operating at 2.0 GHz and contains 256GB physical memory. The interconnect is Mellanox ConnectX-4 EDR 100Gb/s InfiniBand (OFED version 5.0-2.1.8.0) and is configured in a fat-tree topology that is 3:1 oversubscribed. The operating system is CentOS Linux v7.8.2003 (kernel version Linux 3.10.0-1127.19.1.el7.x86_64).

Cluster C : Cascade Lake + InfiniBand : Each node on compute system contains dual socket Intel Xeon Platinum 8280 processors having 28 cores per socket and cores operating at 2.70 GHz speed and contains 192GB of main memory. The interconnect is composed of Mellanox HDR technology (OFED version 5.1-2.5.8) with full HDR (200 Gbps) connectivity between the switches and HDR-100 (100 Gbps) connectivity to the compute nodes. The computing network is configured in a fat-tree topology with a small oversubscription factor of 11:9. The operating system is CentOS Linux release 7.8.2003 (kernel version Linux 3.10.0-1127.19.1.el7.x86_64).

Cluster D : Broadwell + InfiniBand + Quadro RTX 5000 : Each node on system contains dual socket Intel Xeon E5-2620 v4 processors with 16 physical processors operating at 2.10 GHz frequency and equipped with 192 GB DDR4 and 128GB SSD memory. The interconnect is Mellanox

FDR 56Gb/s InfiniBand with OFED version 5.0-2.1.8. The operating system is CentOS Linux v7.8.2003 with kernel version Linux 3.10.0-1127.13.1.el7.x86_64. Four NVIDIA Quadro RTX Turing 5000 GPUs having 16GB GDDR6 memory on each GPU are installed on each node.

Software Configurations : Containers used in this study come from various sources, but very few of them ran without any modification. Sources include repositories from dockerhub [22], [23] and containers built for Cluster C . The software configurations for benchmarks and both applications are listed in Tables II, III and IV.

TABLE II
SOFTWARE CONFIGURATIONS - MPI MICROBENCHMARKS

Cluster	Compiler	MPI	CUDA	Container Platform(s)
Cluster A	GCC 7.3.0	MVAPICH2 GDR 2.3.4	10.2	Singularity 3.5.3
Cluster C	GCC 9.1.0	Intel MPI 19.0.7	-	Charliecloud 0.21 pre

TABLE III
SOFTWARE CONFIGURATIONS - MILC

Cluster	Compiler	MPI	CUDA	Container Platform(s)
Cluster A	GCC 7.3.0	MVAPICH2 GDR 2.3.4	10.2	Singularity 3.5.3
Cluster C	GCC 9.1.0	Intel MPI 19.0.7	-	Charliecloud 0.21 pre

TABLE IV
SOFTWARE CONFIGURATIONS - VPIC

Cluster	Compiler	MPI	CUDA	Container Platform(s)
Cluster A	GCC 7.3.0	MVAPICH2-GDR 2.3.4	10.2	Singularity 3.5.3
Cluster B	Intel 19.0.5	Intel MPI 19.0.5	-	Singularity 3.6.1
Cluster C	Intel 19.1.1	Intel MPI 19.0.7	-	Singularity 3.6.3
Cluster D	GCC 8.3.0	MVAPICH2-GDR 2.3.4	10.1	Singularity 3.6.3

B. Micro-Benchmark Evaluation

We used three MPI benchmarks - MPI_Init, MPI_Bcast and MPI_Alltoall of Intel MPI Benchmarks suite [17] to compare the performance of Singularity and Charliecloud containers with bare metal runs. Each microbenchmark was run at least five times on all the clusters to average out performance variations.

1) Baseline performance: We baseline the performance of containerization with bare metal runs with OSU_Init microbenchmark. Figure 1 plots the time to execute MPI_Init operation at a system scale ranging from 3,584 processes (64 Nodes, 56 PPN) to 229,376 processes(4,096 Nodes, 56 PPN). We observe that container setup and teardown overheads in Charliecloud range between 6% and 14% at various system scales.

2) Collective communication performance: We next establish the overheads of containerization with MPI collective operations. Figure 2a plots the latency of MPI_Bcast operation on 6,144 nodes with 1 process per node (PPN) at Cluster C . We observe on par communication performance by Charliecloud container with bare metal runs at all message sizes. The performance numbers indicate that containerization does not incur any significant performance overheads during runtime, even at a large system scale. To discern setup and teardown overheads of containers with communication collectives, we compare total time to run the MPI Broadcast benchmark at 64, 128, 256, 512, 1K, 2K, 4K, and 6K nodes in Figure 2b. We observe overheads less than 5 seconds to instantiate containers on up to 6,144 nodes at 1 PPN.

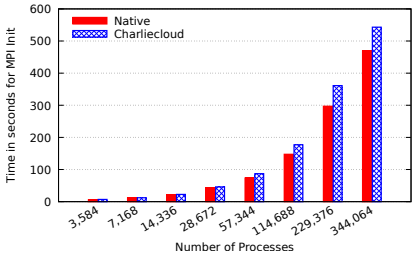
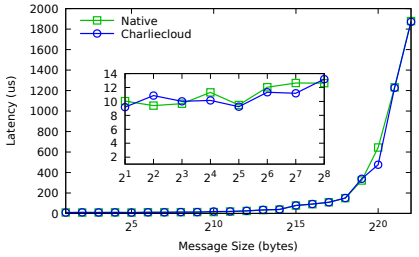
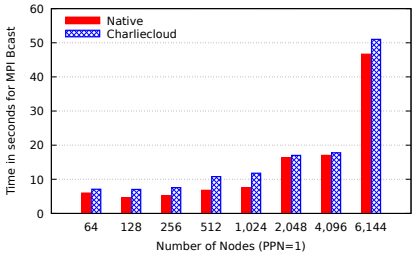


Fig. 1. Baseline performances of containerized and bare metal runs with MPI_Init benchmark on Cluster C



(a) MPI_Bcast

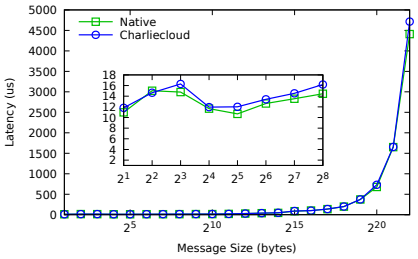


(b) Total Time for MPI_Bcast

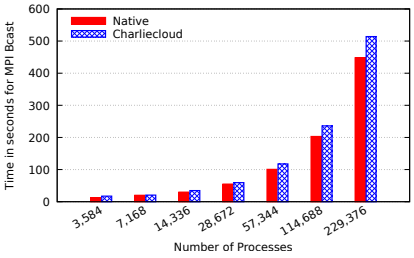
Fig. 2. Performance comparison of Charliecloud containers against native runs with 1 PPN on Cluster C

Since most of the applications in HPC intend to utilize the full potential of computing cores, we, therefore, conducted our next level of evaluations at the full subscription of the nodes. Figure 3a shows the performance of broadcast collective algorithm on 4,096 nodes on Cluster C . Processes per node (PPN) was set to 56 in these experiments which fully subscribe to the nodes on Cluster C . For containerized runs, we instantiated 56 containers through MPI job launcher on each node. We

observe a similar trend in the performance of Charliecloud container and bare metal runs at all the message sizes. Again, to expose the containerization overheads in Charliecloud, we plot the total time to run the complete benchmark at 64, 128, 256, 512, 1K, 2K, and 4K nodes in Figure 3b. We observe an additional 66 seconds of overheads in instantiating the 229,376 containers at 4K nodes. Given the scale at which containers are instantiated, the overheads seem insignificant to run the collective benchmarks.



(a) MPI_Bcast



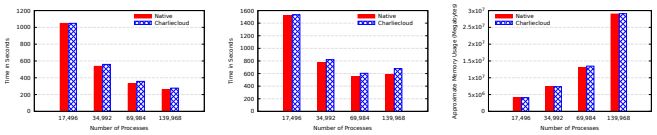
(b) Total Time

Fig. 3. Performance of Charliecloud container against native runs at full subscription of 4,096 nodes on Cluster C

C. Application Level Evaluation

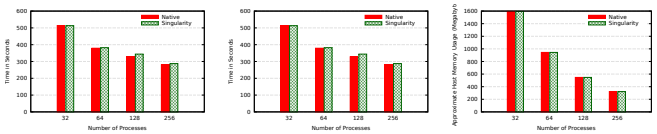
1) MILC: MILC was run with Charliecloud on up to 140K processes at Cluster C and Singularity on up to 256 NVIDIA V100 devices at Cluster A. We set the number of trajectories to one and steps per trajectory to 30. Figure 4 plots the time to solve Conjugate Gradient (CG Time) and Linux time (time command) for native and Charliecloud runs. Charliecloud shows less than 10% overheads at various system scales. Small performance differences are racked up by container instantiation overheads, which is nearly 40 seconds for 0.14 million containers as investigated at the microbenchmark level in Section IV-B. In practice, where MILC is allowed to run for multiple trajectories and several steps per trajectory, the instantiation overheads would become insignificant with long running time of the application. Figure 4c plots the memory consumption reported by the MILC application, which is nearly identical for bare metal and container runs.

On IBM Power 9 Cluster A , MILC was run with the QUDA library to offload computation to NVIDIA V100 devices. From Figure 5, we observe that Singularity incurs less than 4% overheads against bare metal runs. No significant difference in memory consumption was observed at any system size.



(a) CG Time (b) Linux TIME (c) Memory Consumption

Fig. 4. Performance of Charliecloud container against bare metal runs with MILC application on up to 2,592 nodes containing 140K cores on Cluster C



(a) Application Time (b) Linux Time (c) Memory Consumption

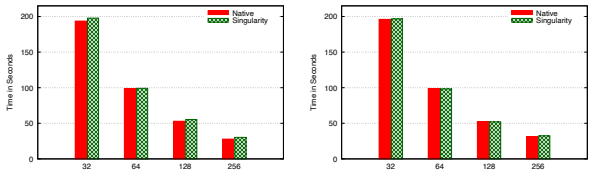
Fig. 5. Performance of Singularity container with MILC application using up to 256 V100 GPUs on Cluster A

Apart from running time and memory usage attributes, we also compare the CPU, device, InfiniBand, NUMA, DRAM, and Lustre parameters and observed on par performance values for all three runtimes. The plots for these attributes are enormous and can be made available on request to the interested researchers.

2) VPIC: The VPIC experiment includes four architectures; two CPU architectures, scaled to 32,768 processes as seen in Figure 7, and two GPU architectures scaled to 256 GPUs as seen in Figure 6. Each experiment is run five times, and the average of the runs are shown in the respective figures. The software used for each experiment is available in Table IV. At each scaling tier, all runs are done within the same job and consequently use the same nodes, fabric location, etc. This is done to reduce the variation associated with running on different nodes and hence network topologies. The authors note this can create a significant discrepancy between “cold” to “warm” cache runs, as the container image and application software and libraries are loaded on a shared parallel filesystem. Although these outliers show a slowdown in first test run within a job whether the test case is bare metal or containerized, they do not show any change in overhead. To combat this, the ”cold cache” outliers are pruned from the averages. Singularity is used as the container platform in order to analyze the overhead of different architectures.

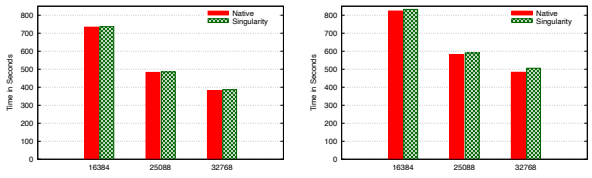
In this experiment, we see in Figures 6 and 7 that architecture does impact the containerization overhead. On average, the RTX platform discrepancy is .29 seconds, while V100 is 2.46 at the same scale (Figures 6). Similarly, on the CPU runs, Rome shows a 3.3 second difference, and Cascade Lake shows the most considerable difference between runs at 13.53 seconds (Figures 7). Although this shows a 4x slowdown, even in the worst case, it is unlikely that the containerization overhead will be impactful for any jobs except those at the largest scale or incredibly short run times.

Our experiments with microbenchmarks and applications indicate that container solutions are an optimal choice for



(a) NVIDIA V100 (b) NVIDIA Quadro RTX 5000

Fig. 6. Performance of Singularity containers against bare-metal runs with VPIC on up to 256 GPUs on Cluster A and Cluster D respectively. (smaller is better)



(a) AMD Rome (b) Intel Cascade Lake

Fig. 7. Performance of Singularity containers against bare-metal runs with VPIC up to 32,768 cores on Cluster B and Cluster C respectively. (smaller is better)

long-running applications. However, short lived applications are benefited from the containers when their build process is complex or time-consuming, and if computing platforms lack required functionalities to run the applications.

V. Conclusion

Recent technological advancements in containerization runtimes have commenced a new trend of HPC software development, which effectively reduces the build and deployment issues caused by complex software dependencies. In this work, we present the challenges of leveraging containerization within HPC systems and showcased the feasibility of two state-of-the-art container technologies. We explore the performance, usability, and portability of container workflows through experiments conducted at a petascale HPC cluster across tens of thousands of processes. We conclude that developers, testers, and end-users can leverage containerization on HPC systems in a performant way, at large scale, to reduce software development and maintenance efforts. The cost of performance at scale is to build support for high-speed interconnects, such as InfiniBand, into the containers. This support does not, however, exclude their use in environments that only have more generic communications support such as TCP/IP or shared memory.

VI. Acknowledgment

This work is supported by UT Austin-Portugal Program, a collaboration between the Portuguese Foundation of Science and Technology and the University of Texas at Austin, award UTA18-001217.

References

[1] D. Merkel, "Docker: Lightweight Linux Containers for Consistent Development and Deployment," *Linux J.*, vol. 2014, no. 239, Mar. 2014.

[2] 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 *IEEE International Conference on Cloud Computing Technology and Science*, 2017.

[3] C. Ruiz, E. Jeanvoine, and L. Nussbaum, "Performance Evaluation of Containers for HPC," in *Euro-Par 2015: Parallel Processing Workshops*, S. Hunold, A. Costan, D. Giménez, A. Iosup, L. Ricci, M. E. Gómez Requena, V. Scarano, A. L. Varbanescu, S. L. Scott, S. Lankes, J. Weidendorfer, and M. Alexander, Eds., 2015.

[4] C. Arango Gutierrez, R. Dernat, and J. Sanabria, "Performance Evaluation of Container-based Virtualization for High Performance Computing Environments," *Revista UIS Ingenierías*, vol. 18, 09 2017.

[5] M. G. Xavier, M. V. Neves, F. D. Rossi, T. C. Ferreto, T. Lange, and C. A. F. De Rose, "Performance Evaluation of Container-Based Virtualization for High Performance Computing Environments," in *PDP'21*, 2013.

[6] D. Brayford and S. Vallecorsa, "Deploying Scientific Al Networks at Petaflop Scale on Secure Large Scale HPC Production Systems with Containers," in *Proceedings of the Platform for Advanced Scientific Computing Conference*, 2020.

[7] Y. Wang, R. T. Evans, and L. Huang, "Performant Container Support for HPC Applications," in *Proceedings of the Practice and Experience in Advanced Research Computing on Rise of the Machines (Learning)*, ser. PEARC '19, 2019.

[8] G. Hu, Y. Zhang, and W. Chen, "Exploring the Performance of Singularity for High Performance Computing Scenarios," in *2019 IEEE 21st International Conference on High Performance Computing and Communications; IEEE 17th International Conference on Smart City; IEEE 5th International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*, 2019.

[9] 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 *IPDPS 2019*, 2019.

[10] 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)*, 2019.

[11] C. Cérin, N. Greneche, and T. Menouer, "Towards Pervasive Containerization of HPC Job Schedulers," in *2020 IEEE 32nd International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, 2020.

[12] R. S. Canon and A. Younge, "A Case for Portability and Reproducibility of HPC Containers," in *2019 IEEE/ACM International Workshop on Containers and New Orchestration Paradigms for Isolated Environments in HPC (CANOPIE-HPC)*, 2019.

[13] N. G. Bachiega, P. S. L. Souza, S. M. Bruschi, and S. d. R. S. de Souza, "Container-Based Performance Evaluation: A Survey and Challenges," in *2018 IEEE International Conference on Cloud Engineering (IC2E)*, 2018.

[14] A. Ruhela, M. Vaughn, S. L. Harrell, G. J. Zynda, J. Fonner, R. T. Evans, and T. Minyard, "Containerization on Petascale HPC Clusters." Texas ScholarWorks, November 2020.

[15] A. Ruhela, M. Vaughn, S. L. Harrell, G. Zynda, J. Fonner, R. T. Evans, and T. Minyard, "Containerization on Petascale HPC Clusters," 2020, in State of Practice track of International Conference for High Performance Computing, Networking, Storage and Analysis (SC20).

[16] "Charliecloud Documentation," https://hpc.github.io/charliecloud/install.html.

[17] "Intel MPI Benchmarks," https://github.com/intel/mpi-benchmarks.

[18] The MIMD Lattice Computation (MILC) Collaboration, www.physics.utah.edu/d̃etar/milc, 2020, Last accessed September 27, 2021.

[19] K. J. Bowers, B. J. Albright, L. Yin, B. Bergen, and T. J. T. Kwan, "Ultrahigh performance three-dimensional electromagnetic relativistic kinetic plasma simulation," *Physics of Plasmas*, 2008.

[20] H. C. Edwards, C. R. Trott, and D. Sunderland, "Kokkos: Enabling manycore performance portability through polymorphic memory access patterns," *Journal of Parallel and Distributed Computing*, vol. 74, 2014.

[21] S. L. Harrell, J. Kitson, R. Bird, S. J. Pennycook, J. Sewall, D. Jacobsen, D. N. Asanza, A. Hsu, H. C. Carrillo, H. Kim, and R. Robey, "Effective Performance Portability," in *2018 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*, 2018.

[22] "Ibmcom/powerai - Docker Hub," https://hub.docker.com/r/ibmcom/powerai/.

[23] "Centos - Docker Hub," https://hub.docker.com/_/centos.