Rootless Containers with Podman for HPC

Holger Gantikow[1,2 [0000−0003−4648−4381]], Steffen Walter[1 [0000−0002−8112−515X]], and Christoph Reich[2 [0000−0001−9831−2181]]

[1] science+computing ag, Atos, Tübingen, DE
gantikow@gmail.com, steffen.walter@atos.net
[2] Institute for Cloud Computing and IT Security, Furtwangen University, Furtwangen, DE
{holger.gantikow,christoph.reich}@hs-furtwangen.de

Abstract. Containers have become popular in HPC environments to improve the mobility of applications and the delivery of user-supplied code. In this paper we evaluate Podman, an enterprise container engine that supports rootless containers, in combination with runc and crun as container runtimes using a real-world workload with LS-DYNA and the industry-standard benchmarks sysbench and STREAM. The results suggest that Podman with crun only introduces a similar low overhead as HPC-focused container technologies.

Keywords: Virtualization · Container · Podman · Singularity · runc · crun · rootless containers · LS-DYNA · Benchmark · Performance Analysis

1 Introduction

Over the last half decade, containers have become a valuable asset when it comes to running services and applications. Especially in enterprise environments, where containers have turned into an integral part of the development, deployment and operation of modern microservice architectures, they are well received. They also prove to be particularly suitable for HPC environments, where they improve the mobility of applications and the delivery of user-supplied code. HPC centers are increasingly confronted with these requirements, since users there are not only demanding software for traditional MPI-based simulations but also increasingly novel software stacks and workflows to support workloads from the data science domain. In addition, modern sites call for solutions that open up ways for simplified use of resources distributed across multiple locations and for obtaining supplementary resources from the cloud [18] at a progressive rate. Containers are a suitable component to support the demands for flexibility in a in many ways converging future of compute. They decouple applications and their dependencies from the underlying operating system and encapsulate them in an easily redistributable unit. Due to special requirements of HPC environments in the form of HPC-specific hardware and related libraries [2] this abstraction is not yet fully realized. Despite these limitations and other open issues [17], the achieved state still represents a serious advancement compared to a few years ago.

At the beginning of the containerization trend started by Docker, there was, as already before with LXC, only one runtime available, which could not be integrated easily into HPC environments due to inherent differences in concepts. As a result, a number

of HPC-focused container runtimes have been developed over the last few years, most notably *Singularity* [10], but also *Shifter* [8], *Charliecloud* [12] and most recently *Sarus* [1]. They made the concept of containers usable in HPC environments, differentiated by diverging functional extent and implementation. However, due to their HPC focus, these runtimes were not widely accepted beyond HPC environments and only Singularity has achieved a notable degree of adoption in the enterprise HPC market. Even though the interoperability of the individual runtimes has well increased due to the *Open Container Initiative (OCI)* specifications, especially enterprise users show an interest in a common solution, which is suitable for a variety of containerized workloads, including HPC, and that consequently also allows a converging of resources.

The *Podman* [4] container engine fills this gap to a certain extent, as it distinguishes itself by features such as the ability to strongly isolate workloads that are particularly relevant in enterprise container environments, as well as an implementation and process model that is much closer to HPC environments. To the best of our knowledge, the potential of Podman in HPC has not been evaluated beyond basic functional testing [13] yet, a gap we are trying to address with our research. The same applies to performance differences between the OCI compatible runtimes *runc* and *crun*, both supported by the Podman engine for spawning and running containers, insights we consider of use for other container engines that rely on on *runc*.

The focus of this paper is to investigate the suitability of Podman in the context of HPC and to identify current limitations. We concentrate on the performance of the container engine, with both *runc* and *crun* as runtime, compared to the native bare metal performance and the HPC-focused Singularity runtime used in enterprise HPC. We are especially interested in the overhead introduced when processing a real-world workload using the *Finite Element Analysis (FEA)* application *LS-DYNA*. This application from the field of *Computer Aided Engineering (CAE)* is widely used in the automotive industry for crash test simulation. This investigation is complemented by a series of industry-standard benchmarks.

The rest of the paper is organized as follows: Section 2 introduces related work on container runtimes targeted at HPC environments with focus on performance overhead. Section 3 covers the basic concepts of Podman in the context of HPC. Section 4 presents the results of our evaluation of the Podman engine with the FEA application LS-DYNA using MPI + Infiniband for communication, as well as CPU and memory performance, using the *sysbench* and *STREAM* benchmarks and gives configuration details related to the environment used for the experimental evaluation. Limitations we came across are discussed in Section 5. The paper concludes in Section 6.

2 Related Work

As Docker was not able to make its mark in HPC centers due to technical challenges, several container platforms were created to address the needs of the HPC community, each characterized by a specific focus and means to implement the privilege escalation required to start containers.

Singularity [10] is characterized by its easy integration into existing HPC workflows, uses a flat single file image format for performance reasons and offers compat-

ibility with legacy OS installations via a setuid starter. *Shifter* [8] reuses some components of the Docker workflow and combines the basic concept of containers with a *chroot* environment. *Charliecloud* [12] is characterized by a very compact code base and the use of user namespaces to spawn containers. *Sarus* [1], the latest HPC-specific runtime, is built around OCI specifications, uses the runtime specification reference implementation *runc* and extends features for HPC use cases by the use of OCI hooks.

The extent of overhead introduced by container virtualization has been investigated many times over the last years. The study carried out by Felter et al. [6] represents the central paper when it comes to Docker containers, as it evaluated a variety of typical services, such as MySQL, in conjunction with standard benchmarks. It concluded that "Docker equals or exceeds KVM performance in every case we tested". This core statement is true throughout various application domains: Di Tommaso et al. [5], who investigated the execution speed of genomic pipelines, shared this conclusion of "negligible impact on the execution performance". Zhang et al. [19] summarized a "much better scalability than virtual machines" with a Spark-based Big Data workload.

These studies have in common that they usually do not consider HPC-optimized runtimes. HPC-specific studies are rather sparse and often only examine partial aspects, such as only comparing a single runtime against native performance as in Wang et al. [16] and Younge et al. [18] - or only non-distributed workloads as in Kovács [9]. The most thorough and recent work is by Torrez et al. [15], where the three HPC-focused runtimes Charliecloud, Shifter, Singularity are compared using a number of standard benchmarks, with the conclusion that "the flexibility gained by using containers does not come at the cost of performance". Sadly the work lacks the inclusion of the Sarus engine and an evaluation of a real-world workload.

3 Podman

Although Docker helped containers achieve their current popularity, it met with disfavor not only in the HPC community, but also in Enterprise Linux distributions. This led to the development of the Podman engine, which integrates more naturally into a Linux system. In order to avoid potential security risks caused by the client-server architecture implemented by Docker, Podman uses a classic fork-exec model, which also improves audit capabilities, as, by lacking user switches, the audit subsystem has the possibility to document which user performed container-related operations. The development of Podman is mainly driven by Red Hat, which leads to the integration of corresponding packages in *Red Hat Enterprise Linux (RHEL)* and its derivatives, a detail that is interesting for HPC environments that often rely on RHEL-based distributions.

Coming from the enterprise breed of container runtimes, Podman follows the "as much isolation as possible" paradigm rather than the "as much isolation as necessary" preferred by HPC container runtimes. As expected, the full range of safeguards for workload isolation is therefore supported. In addition to namespaces and cgroups these include seccomp filters, Linux Security Modules (SELinux, AppArmor) and capabilities. These mechanisms, which we presented in more detail in an earlier work in the context of Docker [7], are not necessarily all implemented by HPC-focused runtimes and may be less relevant to traditional HPC workloads. We expect that with the advent

of a wider variety of applications, converged resources and the need for additional isolation from bare metal operations and concurrent workloads, these will become more important. How we addressed these mechanisms for MPI workloads is documented in Section 4.1.

The most prominent feature of Podman is support for *rootless containers*, which allows the execution of containers without privilege escalation mechanisms, such as root daemon or setuid binary. Podman, like Charliecloud, uses the user namespace functionality for this. Processes inside a new user namespace have different privileges and user IDs than those outside and require corresponding configuration of */etc/sub*{*g,u*}*id*. Limitations of rootless containers are discussed in Section 5.

The Podman engine uses the runtime *runc*, also used by Docker and Sarus, which is written, as Podman, in Go. It also supports the state-of-the-art runtime *crun*, which is implemented in C and described by the developers as “fast and low-memory footprint”. Crun is currently used as the default runtime in Fedora, as it already migrated to the resource limiting feature *cgroups V2*, only supported by crun as of now.

As the name Pod Man(ager) implies, Podman supports the concept of *Pods*, which is used to group a set of containers that collectively implement a complex application. These containers are not completely isolated from each other, but share several namespaces, which simplifies communication. This could be used to provide a group of compute containers with a data sidecar container that only serves the specific input data belonging to a job. In addition, since Kubernetes is based on pods by default, this feature provides increased flexibility in a converged environment to either launch suitable workloads in Kubernetes or to use Kubernetes workloads with Podman.

To build container images Podman relies Buildah and offers the possibility to create OCI compatible images from a Dockerfile without root privileges or background services, which is a notable improvement over Docker-based build processes. It also features functionality to create an image from scratch by using the local package manager to install software as a measure to reduce image bloat over the regular way of running the package manager inside the container itself. It also supports multistage builds, that allow exclusion of build time only tools, as compilers and package managers that are obsolete at run time, from the final build. This can result in smaller image sizes.

A feature that seems more appropriate for HPC environments than for enterprise workloads with stateless containers is the built-in support for *Checkpoint/Restore in Userspace (CRIU)*. It is used to checkpoint containers and restore them at a later time and also to migrate containers to another system, helping with scheduled downtimes and emergency patches. Unfortunately, CRIU cannot be used in combination with rootless containers and when using MPI and Infiniband-based workloads as of now.

Support for OCI Hooks represent a feature we have not yet had the opportunity to gain experience with. OCI Hooks are a mechanism which is used by Sarus to extend the runtime functionality, for example to enable synchronization with the Slurm Scheduler [1]. However, we are aware of an *extended Berkeley Packet Filter (eBPF)*-based hook from the Podman developers, which is used for automated creation of seccomp filters.

4 Evaluation

To investigate the suitability of Podman for HPC workloads we conducted a series of experimental evaluations, comparing rootless Podman to native execution and Singularity as HPC-focused reference runtime. As container runtime for Podman we used both *runc* and *crun* to investigate potential benefits from utilizing the newer implementation written in C (crun) over runtimes in Go (runc, Singularity). This resulted in four distinct runtime environments: *Native*, *Singularity*, *Podman-runc* and *Podman-crun*.

Our test environment is part of an automotive industry environment used for crash test simulations, aero dynamics and other CAE workloads. It consists of one cell with 8 nodes and is part of a cluster with several hundred servers using FDR Infiniband and Gigabit Ethernet as interconnect. The nodes are equipped with SSD-based local storage and use CentOS Linux 8.1 with permissive *SELinux* and a local evaluation user. To ensure meaningful and comparable results all tests were executed on the same systems without configuration changes during the valuation period. Details related to the evaluation environment are documented in Table 1.

Table 1: Evaluation Cluster Details

(a) Hardware Environment

Component	Details
Server	8x NEC HPC 1816Rf-2
Processor	2x Intel Xeon E5-2680 v3 (Hyper Threading enabled)
Memory	128 GB DDR4 (2134 MHz)
Interconnect	FDR Mellanox Infiniband HCA MT27500 (ConnectX-3)
Local Storage	Intel SSDSC2BB80

(b) Software Environment

Component	Version
Operating System	CentOS Linux 8.1.1911
Kernel	4.18.0-147.3.1.el8_1.x86_64
Singularity	3.5.2
Podman	1.6.4
crun	0.13
Platform MPI	9.1.4.3r
LS-DYNA	mpp_r9_3_dm_134916

As real-world workload we use two versions of a LS-DYNA crash simulation, the core application the cluster is used with on a daily basis.

To achieve a more detailed view of the overhead induced by containers, we also perform a series of industry-standard benchmarks. To measure *CPU performance* we utilize *sysbench* that calculates the prime numbers between 1 and 40 million using one thread per CPU core. For *memory performance* we rely on STREAM [11]. Benchmark versions, configurations and execution calls are documented in Table 2.

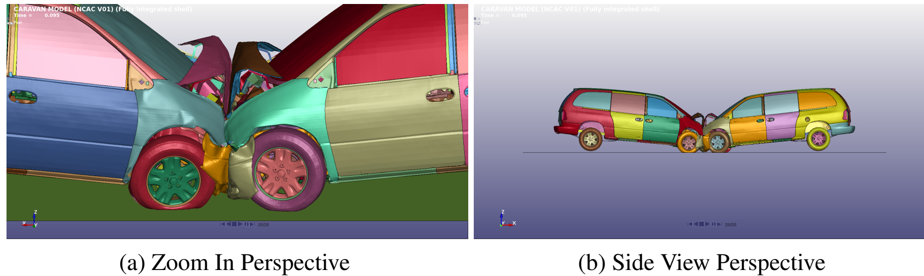
4.1 LS-DYNA

LS-DYNA is used to simulate the impact of automotive crashes, explosions and sheet metal stamping. We use the *car2car* model, a widely used workload originally published by the *National Crash Analysis Center (NCAC)* to measure the performance of LS-DYNA. The model includes 2.4 million single elements and simulates a frontal

Table 2: Benchmark Configuration

Component	Version	Compile Flags, Call, Configuration
STREAM	5.10	*gcc -m64 -O3 -mcmodel=medium -ffreestanding -fopenmp -DSTREAM_ARRAY_SIZE=3000000000 -DSTREAM=double stream.c -o stream*
sysbench CPU	1.0.17	*sysbench cpu –threads=24 –cpu-max-prime=40000000 run*
LS-DYNA car2car short	V03c	*memory=600m memory2=60m endtime=0.02*
LS-DYNA car2car long	V03c	*memory=600m memory2=60m [default endtime=0.12]*

crash of of two minivans (see Figure 1) each at the speed of 35 mph and covers the first 120 ms of the crash. For quicker turnaround times we created an additional model that only covers the first 20 ms. We refer to these workloads in the course of this paper as *short run* (20 ms) and *long run* (120 ms) configuration.



(a) Zoom In Perspective (b) Side View Perspective

Fig. 1: NCAC car2car Model at time = 95 ms

We used LS-DYNA on 8 nodes with 24 processes each, as it does not scale very well horizontally and the selected configuration has proven to be efficient. The rest of the configuration is characterized as follows: All processes on one host share a working directory, that is located on a local SSD. The inter-process communication on each host is accomplished via shared memory and the inter-node communication is implemented by remote memory access via FDR Infiniband. Upon using *mpirun* to start a containerized run, one container is being created for each process. The different jobs were directly started with *mpirun*, omitting batch system integration, but still maintaining an identical LS-DYNA invocation throughout the tests. We extended the shell environment with information regarding the LS-DYNA licence server, *Platform MPI* library path and MPI meta variables, and passed the updated environment into the container that starts the application. To obtain meaningful results, each LS-DYNA run was carried out six times with each configuration in every runtime environment.

Starting the containerized workload with Singularity was straightforward and required no additional parameters or adjustments: *mpirun mpp i=Caravan-V03c-2400k-main-shell16-120ms.k memory=600 memory2=60*.

Since Podman is not developed specifically for HPC workloads, we had to make a some adjustments to the container execution call: interprocess communication and

shared memory access require *pid* and *ipc* namespace sharing among the containers. The *net* namespace must also be shared between the containers and the host, so that MPI on the host can manage internode communication. These adjustments are achieved by appending the command line parameters *–net=host –pid=host –ipc=host*. Furthermore we used *–env=host* to pass the host environment into the container and *–volume* to bind mount a shared working directory and pass the Infiniband device into the containers.

Table 3: LS-DYNA: Arithmetic mean + overhead

	Native	**Singularity**	**Podman-runc**	**Podman-crun**
Short Run	1097,67 s	1116,17 s	1154,83 s	1120,17s
Long Run	6393,33 s	6521,17 s	6712,00 s	6521,83s
Overhead Short	-	2,09%	5,63%	2,45%
Overhead Long	-	1,61%	4,58%	1,62%
Mean Overhead	-	1,85%	5,10%	2,04%

The analysis of the results [3] (see Figure 2 and Table 3) from the different runs show: **a)** All container runtimes introduce a certain amount of overhead compared to native execution of the simulation (1,85% - 5,10% for the arithmetic mean of short and long run). **b)** This overhead might be negligible for many workloads over the benefits of containers, as 2/3 runtimes only add 2,04% or less overhead. **c)** Although Singularity causes the least overhead, the differences of Podman in combination with crun as runtime are minor, especially for long runs of LS-DYNA (1,61% vs 1,62%). **d)** Compared to Singularity and crun the performance of runc is noticeably lower.

Discussions with Podman developers indicate that the slight difference of Podman with *crun* compared to Singularity might be related to Podman isolation mechanisms activated by default, such as a seccomp profile or more extensive use of namespaces.

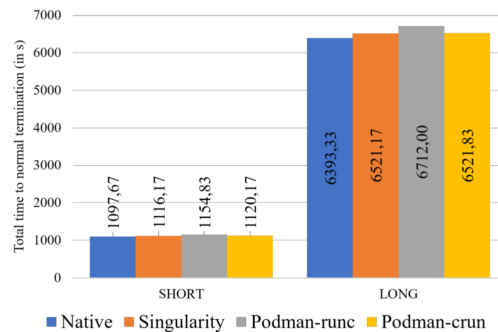


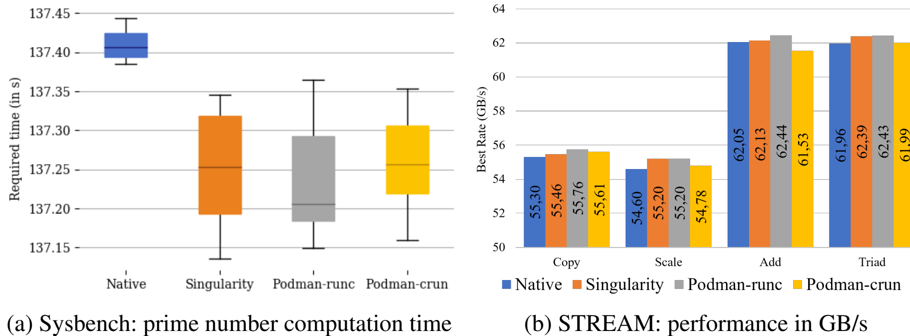
Fig. 2: LS-DYNA: Elapsed time to normal termination for car2car model

[3] Values used in Figure 2 and Table 3 are based on *"elapsed time"* as returned by LS-DYNA and do not include startup and teardown periods, which are further discussed Section 5.

4.2 Benchmarks

To ensure comparability with other evaluations and to get specifics on where the overhead described in Section 4.1 comes from, we performed some additional industry-standard benchmarks on a single node. We used sysbench and STREAM to measure CPU and memory performance. *Sysbench* is a multi-threaded benchmark tool that supports different tests to measure performance. The evaluation scope of this paper covers the CPU test. *STREAM* performs simple vector operations and measures the available memory bandwidth. It needs to be compiled to use a data set that is significantly larger than the CPU caches. To reflect CPU cache size and the amount of memory available in our test systems we modified the preprocessor definition *STREAM_ARRAY_SIZE* to increase the elements per array to 3 billion. When using large data sets, the resulting values are naturally dominated by the bandwidth between the CPU and the memory rather than the handling of cache misses [11]. Details concerning benchmark versions, configurations and execution calls are documented in Table 2.

Figure 3a shows the results of the sysbench CPU benchmark that were sampled over 25 runs with each runtime environment. The variation of the execution times reported by the benchmark reported is $\approx$ 300 ms, not counting rare outliers. Therefore, the results only differ by 0.4%, which does not lead to any direct conclusions, except that the results of all container-based runtime environments are virtually identical and there seems to be no added overhead by containerization.



(a) Sysbench: prime number computation time

(b) STREAM: performance in GB/s

Fig. 3: Sysbench + STREAM results

Figure 3b illustrates the arithmetic mean of the performance data collected during five runs of the STREAM benchmark for STREAM's four memory-bound vector kernels in each runtime environment. Again, the measurements are within a very small window, the results differ by less than 1% and containerization does not seem to impose additional overhead in most cases.

The *Copy* part of STREAM, which only copies the elements of one vector into another ($C[i] = A[i] | i = 1 \dots n$ [11]), shows similarities to the results of the LS-DYNA benchmark. It can be concluded that the overhead of containerized LS-DYNA is at least partially due to memory intensive operations.

5 Limitations

During our evaluation we were affected by limitations of *rootless* Podman: Some Podman commands fail when no subordinate user IDs (subuid) and subordinate group IDs (subgid) are not configured, even though we did not make use of the feature since sub{u,g}ids cannot be used with MPI jobs, since shared memory access and shared file access is not supported.

When using network based authentication systems like LDAP, sub{u,g}ids cannot be configured easily, because the files /etc/sub{u,g}id are not utilized. According to Podman Developers the addition of new directives in the nsswitch.conf to support the *subuid* and *subgid* databases is work in progress. In rare cases when trying to start new containers for a job run, it failed with an error indicating that there already exists a container storage for the requested name, even though none by that name is known to Podman at the time.

We observed that startup and teardown costs of Podman containers are considerably greater than that of Singularity containers. We measured the following deltas when comparing the total *real* wall clock from the start of *mpirun* until successful termination and the *"elapsed time"* as reported by LS-DYNA: native: $\approx$ 3 s, Singularity: $\approx$ 5 s, Podman: $\approx$ 25 s. We are currently working with the developers to clarify the cause, but the most probable reason is due to the layered filesystem that needs to be set up and torn down again in combination with a $\approx$ 2GB LS-DYNA image.

Other limitations that did not affect our evaluation, but can be relevant to other HPC environments, are the lack of direct Slurm support, failure to bind to ports < 1024 (privileged ports), and the inability to run from home directories on NFS or GPFS. In addition, rootless Podman does not support CRIU's checkpoint and restore features and cgroups V1 yet. The latter issue will be solved in the long run by moving to V2, which is already supported in Fedora 31 for crun [14]. A comprehensive list of limitations that apply to rootless containers is maintained by the Podman developers [3].

6 Conclusion

Our evaluation showed that Podman, despite the different focus of the project, is essentially suitable for use in HPC. In terms of real-world workload performance, Podman performs on a similar level as Singularity, at least in conjunction with crun as runtime. Our results with industry-standard benchmarks are consistent with other studies on other runtimes, namely that containers generate only a small performance overhead, if at all. However, it is still higher with real-world workloads than with benchmarks. At the moment there are still some limitations in Podman, including issues that should be fixed in future versions, but which cause more problems in a production environment than in our evaluation environment. From an administrator's point of view, these include restrictions on the use of directory services such as LDAP, shortcoming of rootless containers in combination with distributed file systems and, from the user's point of view, a more complex integration with MPI workloads than Singularity. Nevertheless, Podman offers a number of advantageous features. These include the ability to create images with user privileges only, support for pods and stronger isolation. This makes it

an interesting option for newer workloads and converged environments. In future work we want examine Podman specific features and explore the possibilities of OCI Hooks, especially for integration with workload managers.

References

1. Benedicic, L., Cruz, F.A., Madonna, A., Mariotti, K.: Sarus: Highly Scalable Docker Containers for HPC Systems. In: High Performance Computing. pp. 463–477. Springer International Publishing (2019)
2. Canon, R.S., Younge, A.: A Case for Portability and Reproducibility of HPC Containers. pp. 49–54. IEEE (2019)
3. Containers Organization: libpod: Shortcomings of Rootless Podman, https://github.com/containers/libpod/blob/master/rootless.md
4. Containers Organization: Podman — podman.io, https://podman.io/
5. Di Tommaso, P., Palumbo, E., Chatzou, M., Prieto, P., Heuer, M.L., Notredame, C.: The impact of Docker containers on the performance of genomic pipelines. PeerJ **3**, e1273 (2015)
6. Felter, W., Ferreira, A., Rajamony, R., Rubio, J.: An updated performance comparison of virtual machines and linux containers. In: 2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS). pp. 171–172 (March 2015)
7. Gantikow, H., Reich, C., Knahl, M., Clarke, N.: Providing security in container-based HPC runtime environments. Lecture Notes in Computer Science pp. 685–695 (2016)
8. Jacobsen, D.M., Canon, R.S.: Contain This, Unleashing Docker for HPC. Cray User Group 2015 p. 14 (2015), https://www.nersc.gov/assets/Uploads/cug2015udi.pdf
9. Kovács, Á.: Comparison of different linux containers. In: 2017 40th International Conference on Telecommunications and Signal Processing (TSP). pp. 47–51. IEEE (2017)
10. Kurtzer, G.M., Sochat, V., Bauer, M.W.: Singularity: Scientific containers for mobility of compute. PLOS ONE **12**(5), 1–20 (05 2017)
11. McCalpin, J.D.: Memory Bandwidth and Machine Balance in Current High Performance Computers. IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter (May), 19–25 (1995)
12. Priedhorsky, R., Randles, T.: Charliecloud: Unprivileged Containers for User-Defined Software Stacks in HPC. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. SC 17, Association for Computing Machinery, New York, NY, USA (2017)
13. Reber, A.: Podman in HPC environments, https://podman.io/blogs/2019/09/26/podman-in-hpc.html
14. Suda, A.: The current adoption status of cgroup v2 in containers, https://medium.com/nttlabs/cgroup-v2-596d035be4d7
15. Torrez, A., Randles, T., Priedhorsky, R.: HPC Container Runtimes have Minimal or No Performance Impact. pp. 37–42. IEEE (2019)
16. Wang, Y., Evans, R.T., Huang, L.: Performant container support for HPC applications. In: ACM International Conference Proceeding Series (2019)
17. Watada, J., Roy, A., Kadikar, R., Pham, H., Xu, B.: Emerging trends, techniques and open issues of containerization: A review. IEEE Access **7**, 152443–152472 (2019)
18. Younge, A.J., Pedretti, K., Grant, R.E., Brightwell, R.: A Tale of Two Systems: Using Containers to Deploy HPC Applications on Supercomputers and Clouds. In: Proceedings of the International Conference on Cloud Computing Technology and Science, CloudCom (2017)
19. Zhang, Q., Liu, L., Pu, C., Dou, Q., Wu, L., Zhou, W.: A Comparative Study of Containers and Virtual Machines in Big Data Environment. In: IEEE International Conference on Cloud Computing, CLOUD. vol. 2018-July, pp. 178–185 (2018)