Sarus: Highly Scalable Docker Containers for HPC Systems

Lucas Benedicic(✉), Felipe A. Cruz, Alberto Madonna, and Kean Mariotti

Swiss National Supercomputing Centre, Lugano, Switzerland
benedicic@cscs.ch

Abstract. The convergence of HPC and cloud computing is pushing HPC service providers to enrich their service portfolio with workflows based on complex software stacks. Such transformation presents an opportunity for the science community to improve its computing practices with solutions developed in enterprise environments. Software containers increase productivity by packaging applications into portable units that are easy to deploy, but generally come at the expense of performance and scalability. This work presents Sarus, a container engine for HPC environments that offers security oriented to multi-tenant systems, container filesystems tailored for parallel storage, compatibility with Docker images, user-scoped image management, and integration with workload managers. Docker containers of HPC applications deployed with Sarus on up to 2888 GPU nodes show two significant results: OCI hooks allow users and system administrators to transparently benefit from plugins that enable system-specific hardware; and the same level of performance and scalability than native execution is achieved.

1 Introduction

Building and deploying software on high-end computing systems is a challenging task. Recent service models developed for cloud environments are putting additional pressure on the traditional support and maintenance methods of HPC centers. Even regular upgrades of HPC software environments have become more demanding in the presence of applications with shorter release cycles (e.g. TensorFlow [1], Spark [42]). Increasingly complex software stacks are required by certain workflows like Jupyter notebooks, visualization and data analysis. Consequently, finding a mechanism that helps lowering the support burden coming from end users as well as from regular maintenance tasks, while still delivering a rich palette of applications and services running at high performance, is becoming a matter of greater importance for HPC service providers. The progressive integration of cloud computing usage models into HPC should not be viewed as a challenge, but rather as an opportunity to improve existing and future HPC capabilities.

The original version of this chapter was revised: It has been changed to open access under a CC BY 4.0 license and the copyright holder is now "The Author(s)". The correction to this chapter is available at https://doi.org/10.1007/978-3-030-34356-9_50

M. Weiland et al. (Eds.): ISC 2019 Workshops, LNCS 11887, pp. 46–60, 2019.
https://doi.org/10.1007/978-3-030-34356-9_5

One such example, which received significant development momentum from cloud services, is represented by software containers, a virtualization technology that can be employed to cope with the requirements of flexible yet increasingly sophisticated application deployments. By packaging applications and their environments into standard units of software, containers feature several benefits: they are portable, easy to build and deploy, they have a small footprint, and also show a low runtime overhead. The portability of containers is achieved through a runtime program that performs the instantiation of images which are agnostic of the host infrastructure [29].

Ensuring their performance and scalability matches native deployments as close as possible is critical to the success of containers in an HPC context. This has a direct impact on particular elements of a container image, such as support for dedicated hardware, drivers and vendor-specific libraries which are usually not part of the image itself due to their emphasis on portability.

Another key aspect to take into consideration is to leverage on existing developments. The technology provided by Docker established itself as the most popular container implementation. For this reason, a container software for HPC providing the capability to deploy Docker-compatible containers at scale will greatly benefit from a rich set of mature tools and practices.

Based on our experience operating containers in an HPC production environment, we identify the following key features for the effective use of containers in HPC:

(F1) Transparently achieve native performance and scalability from specialized hardware at runtime.
(F2) Suitability for parallel filesystems.
(F3) Security measures fitting multi-tenant systems.
(F4) Compatibility with workload managers.
(F5) Easy integration of vendor support.
(F6) Compatibility with Docker's image format and workflow.

The rest of this paper is organized as follows: in Sect. 2 we introduce related work of container runtimes targeted at HPC environments. Section 3 introduces Sarus, describing design and implementation details. In Sect. 4 we illustrate five OCI hooks to satisfy multiple HPC use cases and show how Sarus supports them. In Sect. 5 we proceed to test the capabilities of Sarus by comparing the performance of containerized and native versions from a selection of real-world HPC applications. These tests also demonstrate the use of OCI hooks to provide access to high-performance MPI and GPU acceleration, making full use of the hardware available on a supercomputing system.

2 Related Work

In the scientific computing community there has been a growing interest in running Docker containers on HPC infrastructure [8]. However, Docker was developed to answer the particular needs of web service applications, which feature substantially different workloads from the ones typically found in HPC. For

this reason, deploying Docker in a production HPC environment encounters significant technical challenges, like missing integration with workload managers, missing support for diskless nodes, no support for kernel-bypassing devices (e.g. accelerators and NICs), no adequate parallel storage driver, and a security model unfit for multi-tenant systems.

A number of container platforms have thus been created to fill this gap and provide Linux containers according to the demands of HPC practitioners, retaining a varying degree of compatibility with Docker.

Singularity [19] is a container platform designed around an image format consisting of a single file. While it is able to convert Docker images through an import mechanism, it is centered on its own non-standard format and can also build its own images starting from a custom language. Support for specific features, e.g. OpenMPI and NVIDIA GPUs, is integrated into the Singularity codebase.

Charliecloud [33] is an extremely lightweight container solution focusing on unprivileged containers. It is able to import Docker images, requires no configuration effort from system administrator, and it is a user-space application, achieving high levels of security. Charliecloud's minimal feature set allows fine-grained control over the customization of the container. On the other hand, it requires a substantial level of proficiency from the user to set up containers for non-trivial use cases.

Shifter [16] is a software developed to run containers based on Docker images on HPC infrastructures. It is designed to integrate with the Docker workflow and features an image gateway service to pull images from Docker registries. Shifter has recently introduced the use of configurable software modules for improved flexibility. However, the modules and the executables have to be developed and configured specifically for Shifter.

3 Sarus

Sarus is a tool for HPC systems that instantiates feature-rich containers from Docker images. It fully manages the container life-cycle by: pulling images from registries, managing image storage, creating the container root filesystem, and configuring hooks and namespaces. For instantiating the containers, Sarus leverages on `runc` [30]: an OCI-compliant kernel-level tool that was originally developed by Docker [22].

In order to address the unique requirements of HPC installations, Sarus extends the capabilities of `runc` to enable features such as native performance from dedicated hardware, improved security on multi-tenant systems, support for network parallel filesystems, diskless computing nodes, and compatibility with workload managers. Keeping flexibility, extensibility, and community efforts in high regard, Sarus relies on industry standards and open source software.

Similarly, Sarus depends on a widely-used set of libraries, tools, and technologies to reap several benefits: reduce maintenance effort and lower the entry barrier for service providers wishing to install the software, and for developers

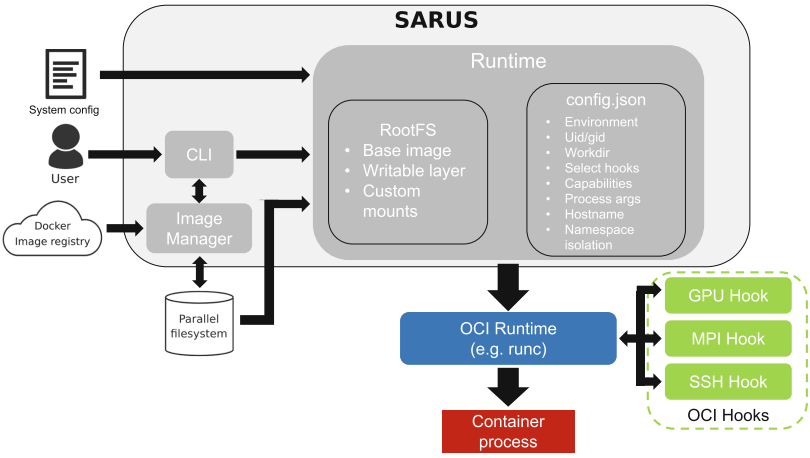

Fig. 1. Sarus architecture diagram.

seeking to contribute code. Sarus builds upon the popularity of Docker, providing the capability of deploying Docker images, the internal structure of which is based on OCI standards.

3.1 Sarus Architecture

The workflows supported by Sarus are implemented through the interaction of several software components (Fig. 1). The **CLI** component processes the command line arguments and calls other components to perform the actions requested by the user. The **Image Manager** component is responsible for importing container images onto the system, converting the images to Sarus' own format, storing them on local system repositories, and querying the contents of such repositories. The **Runtime** component instantiates and executes containers by setting up a bundle according to the OCI Runtime Specification: such bundle is made of a root filesystem directory for the container and a JSON configuration file. After preparing the bundle, the **Runtime** component will call an external **OCI runtime** (`runc` in this case), which will effectively spawn the container process. Sarus can instruct the **OCI runtime** to use one or more **OCI hooks**. These hooks are often used to customize the container by performing actions at selected points during the container lifetime.

3.2 Container Creation

The Runtime component of Sarus is responsible for setting up and coordinating the launch of container instances. When the user requests the execution of a container process through the `sarus run` command, an OCI bundle is first created in a dedicated directory. The bundle is formed by a *rootfs* directory, containing

the root filesystem for the container, and a *config.json* file providing settings to the OCI runtime.

Before actually generating the contents of the bundle, Sarus will create and join a new Linux mount namespace, then mount an in-memory temporary filesystem on the designated OCI bundle directory. This procedure prevents other processes of the host system from having visibility on any artifact related to the container instance [25,26] (F3) and ensures complete cleanup of container resources upon termination.

In the next subsections, we will describe the generation of the bundle contents in more detail.

Root Filesystem. The root filesystem for the container is assembled in a dedicated directory inside the OCI bundle location through several steps:

a. The squashfs file corresponding to the image requested by the user is mounted as a *loop device* on the configured rootfs mount point. The loop mount allows access to the image filesystem as if it resided on a real block device (i.e. a storage drive). This strategy prevents metadata thrashing and improves caching behavior (F2), as all container instances access a single squashfs file on the parallel filesystem. The effectiveness of this approach has already been demonstrated by Shifter [16].
b. Sarus proceeds to create an overlay filesystem [5], using the loop-mounted image as the *read-only* lower layer, while part of the OCI bundle in-memory filesystem forms the *writable* upper layer. An overlay filesystem allows the contents of containers to be transparently modifiable by the users, while preserving the integrity of container images (F6). Additionally, the in-memory filesystem improves the performance of the container writable layer and suits diskless computing nodes (e.g. as those found in Cray XC systems), where the host filesystem also resides in RAM.
c. Selected system configuration files (e.g. `/etc/hosts`, `/etc/passwd`, `/etc/group`) required to setup file permissions in shared directories, or networking with other computing nodes, are copied from the host into the rootfs of the container (F3).
d. *Custom mounts* are performed. These are bind mounts requested by the system administrator or by the user to customize the container according to the needs and resources of an HPC system site or a specific use case, such as providing access to parallel filesystems.
e. The container's rootfs is remounted [43] to remove potential suid bits from all its files and directories (F3).

config.json. The JSON configuration file of the OCI bundle is generated by combining data from the runtime execution context, command-line parameters, and properties coming from the image. We hereby highlight the most important details:

- The uid/gid of the user from the host system are assigned to the container process, regardless of the user settings in the original image. This is done to keep a consistent experience with the host system, especially regarding file ownership and access permissions (F3).
- The container environment variables are created by uniting the variables from the host environment and the variables from the image. If a variable exists in both the host and the image, the value from the image is taken. This ensures a consistent behaviour as expected by image creators (e.g. in the case of `PATH`). Selected variables are also adapted by Sarus to suit system-specific extensions, like NVIDIA GPU support, native MPI support or container SSH connections (F1, F4, F5, F6).
- Support for image-defined properties like default arguments, entrypoint, and working directory is configured to show consistent behavior with a Docker container (F6).
- The container process is configured to run with all Linux capabilities disabled[1] and it is prevented from acquiring new privileges by any means[2] (F3).
- Settings for OCI hooks are copied from the Sarus configuration file.

4 Extending Sarus with OCI Hooks

OCI hooks are an effective way of extending the functionality provided by a container runtime. The standard interface defined by the OCI Runtime Specification allows the extensions to be developed without understanding the details about how the runtime instantiates a container. This way, vendors can independently develop dedicated hooks to provide support for their products (F5). Likewise, researchers and engineers at computing sites can create additional hooks for enabling innovative capabilities or further integrate containers with their HPC infrastructure. Even hooks developed by the open source community or meant for other areas of IT can be seamlessly and readily adopted by an OCI-compliant runtime.

The nature of hooks as autonomous extensions implies that each one of them can be activated independently from each other. Sarus gives system administrators full flexibility to configure hooks, so installations can be tailored to the characteristic of each individual system.

In the context of HPC, hooks have shown the potential to augment containers based on open standards, including native support for dedicated hardware like accelerators and interconnect technologies (see Sect. 5) (F1).

In the next subsections we highlight several OCI hooks use cases of particular interest for HPC.

[1] Linux divides the privileges traditionally associated with superuser into distinct units, known as capabilities [24].

[2] This is achieved by setting the *no_new_privs* process attribute to 1 [22].

4.1 Native MPICH-Based MPI Support (H1)

The MPI [11] is a portable standard for developing distributed memory applications. HPC centers around the globe extensively deploy MPI implementations that can take advantage of the fast network infrastructure while providing message-passing communications that are efficient and scalable.

In 2013 multiple vendors of MPI implementations based on the MPICH library announced an effort to achieve ABI compatibility between their implementations [27]. In practice, applications that have been built with any library complying with the MPICH ABI Compatibility Initiative can later run correctly using other ABI initiative libraries without needing recompilation.

Sarus features a hook able to swap MPICH-based MPI implementations inside a container with an ABI-compatible native implementation from the host. The detection of host MPI libraries is performed through the `ldconfig` system tool [23]. The advantage is that the native library has been optimized for the infrastructure of the HPC system, which is able to leverage hardware acceleration, therefore allowing the container to achieve native performance (F1). ABI compatibility between host and container MPI libraries is fundamental to ensure the replacement works seamlessly at runtime without requiring a recompilation of the application.

4.2 NVIDIA GPU Support (H2)

The use of NVIDIA GPUs as computational accelerators for HPC has become increasingly relevant in recent times. The November 2018 edition of the TOP500 list of supercomputer sites features 123 systems equipped with NVIDIA GPUs, including the top 2 systems [39]. Additionally, several GPU-accelerated codes were recognized as finalists or winners of the ACM Gordon Bell Prize over the years [17,18,37,40].

NVIDIA provides access to GPU devices and their drivers inside OCI containers through the hook component of the NVIDIA Container Runtime [28]. The hook imports the native device files and driver stack into the container, using `ldconfig` to find host shared libraries.

Sarus is able to integrate the use of the NVIDIA Container Runtime hook by modifying the container environment in a way that is completely transparent to both users and system administrators (F1, F5). These modifications ensure that device allocations performed by the workload manager are respected, while guaranteeing the correct operation of CUDA applications inside the container, even in the case of partial or shuffled devices selection on multi-GPU systems (F4). For example, when operating under a workload manager which sets the `CUDA_VISIBLE_DEVICES` environment variable, no user action is required to correctly enable access to GPUs from inside the container.

This is also the first example of a vendor-supplied OCI hook that Sarus is able to seamlessly integrate and apply to high-performance containers.

4.3 SSH Connection Within Containers (H3)

The SSH protocol [41] is used to securely connect machines and services over a network. Such capability is required by distributed applications like Apache Spark [42].

Sarus provides a hook that gives containers the ability to establish SSH connections between them, using a customized OpenSSH software [12], a set of dedicated keys (avoiding reuse of the user's native keys for security reasons), and a non-standard port.

4.4 Slurm Scheduler Synchronization (H4)

The Slurm Workload Manager [35] is an open-source job scheduler adopted by many high performance systems worldwide to allocate system resources in order to carry out different computing jobs. In some cases, the allocated resources are not all made available at the exact same time.

Sarus comes with a hook that implements a synchronization barrier before the user application is started inside the container. This prevents actions (such as attempting SSH connections) towards containers which are not yet available from causing the whole Slurm job step to fail. The synchronization mechanism is based on the environment set up by Slurm with regards to total number of tasks and process identifiers.

5 Performance Evaluation

In this section we discuss the performance aspects of Sarus by comparing data for a variety of workloads, executed by corresponding native and containerized applications. It should be noted that the creation of the container images represents a best reproducibility effort in terms of software releases, compiler versions, compilation tool chains, compilation options and libraries, between native and containerized versions of the applications. Despite this effort, the exact reproducibility of the application binaries cannot be guaranteed. However, this highlights how the Docker workflow of packaging applications using mainstream Linux distributions (e.g. Ubuntu) and basic system tools (e.g. package managers) can seamlessly integrate with HPC environments, since container images targeted at personal workstations can still be used to achieve consistently comparable performance with native installations.

For each data point, we present the average and standard deviation of 50 runs, to produce statistically relevant results, unless otherwise noted. For a given application, all repetitions at each node count for both native and container execution were performed on the same allocated set of nodes.

We conduct our tests on Piz Daint, a hybrid Cray XC50/XC40 system in production at the Swiss National Supercomputing Centre (CSCS) in Lugano, Switzerland. The compute nodes are connected by the Cray Aries interconnect under a Dragonfly topology, providing users access to hybrid CPU-GPU nodes.

Hybrid nodes are equipped with an Intel® Xeon® E5-2690v3 processor, 64 GB of RAM, and a single NVIDIA® Tesla® P100 with 16 GB of memory. The software environment on Piz Daint is the Cray Linux Environment 6.0.UP07 (CLE 6.0) [7] using *Environment Modules* [9] to provide access to compilers, tools, and applications. The default versions for the NVIDIA CUDA and MPI software stacks are, respectively, CUDA version 9.1, and Cray MPT version 7.7.2.

We install and configure Sarus on Piz Daint to use `runc`, the native MPI (H1) and NVIDIA Container Runtime (H2) hooks introduced in Sect. 4, and to mount container images from a Lustre parallel filesystem [31].

5.1 Scientific Applications

In this section, we test three popular scientific application frameworks, widely used in both research and industry. The objective is to demonstrate the capability of Sarus and its HPC extensions (see Sect. 4) to run real-world production workloads using containers, while performing on par with highly-tuned native deployments. All container runs in the following subsections use both the MPI and NVIDIA GPU hooks.

GROMACS. GROMACS [2] is a molecular dynamics package with an extensive array of modeling, simulation and analysis capabilities. While primarily developed for the simulation of biochemical molecules, its broad adoption includes research fields such as non-biological chemistry, metadynamics and mesoscale physics.

For the experiment we select the GROMACS Test Case B from PRACE's Unified European Applications Benchmark Suite [32]. The test case consists of a model of cellulose and lignocellulosic biomass in an aqueous solution. This inhomogeneous system of 3.3 million atoms uses reaction-field electrostatics instead of smooth particle-mesh Ewald (SPME) [10], and therefore should scale well. The simulation was carried out using single precision, 1 MPI process per node and 12 OpenMP threads per MPI process. We perform runs from a minimum of 4 nodes up to 256 nodes, increasing the node count in powers of two, carrying out 40 repetitions for each data point.

As the native application we use GROMACS release 2018.3, built by CSCS staff and available on Piz Daint through an environment module. For the container application, we build GROMACS 2018.3 with GPU acceleration inside a Ubuntu-based container.

The results are illustrated in Fig. 2. We measure performance in ns/day as reported by the application logs. The speedup values are computed using the performance average of each data point, taking the native value at 4 nodes as baseline.

We observe the container application consistently matching the scalability profile of native version. Absolute performance is identical up to 32 nodes. From 64 nodes upwards, the differences (up to 2.7% at 256 nodes) are consistent with empirical experience about sharing the system with other users during

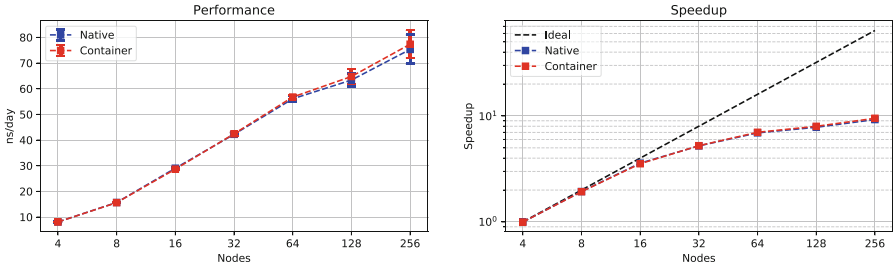

Fig. 2. Comparison of performance and speedup between native and container versions of GROMACS on Piz Daint.

the experiment. Standard deviation values remain comparable across all node counts.

TensorFlow with Horovod. TensorFlow [1] is a popular open source software framework for the development of machine-learning (ML) models, with a focus on deep neural networks.

Horovod [36] is a framework to perform distributed training of deep neural networks on top of another ML framework, like TensorFlow, Keras, or PyTorch. Notably, it allows to replace TensorFlow's own parameter server architecture for distributed training with communications based on an MPI model, providing improved usability and performance.

As test case, we select the `tf_cnn_benchmark` scripts from the Tensorflow project [38] for benchmarking convolutional neural networks. We use a ResNet-50 model [14] with a batch size of 64 and the synthetic image data which the benchmark scripts are able to generate autonomously. We perform runs from a minimum of 2 nodes up to 512, increasing the node count in powers of two.

For the native application, we install Horovod 0.15.1 on Piz Daint on top of a CSCS-provided build of TensorFlow 1.7.0, which is available on Piz Daint through an environment module. For the container application, we customize the Dockerfile provided by Horovod for version 0.15.1, which is based on Ubuntu, to use TensorFlow 1.7.0 and MPICH 3.1.4. Neither application uses NVIDIA's NCCL library for any MPI operation.

The results are shown in Fig. 3. We measure performance in images per second as reported by the application logs and compute speedup values using the performance averages for each data point, taking the native performance at 2 nodes as baseline.

The container application shows a performance trend identical to the native one, with both versions maintaining close standard deviation values. Absolute performance differences up to 8 nodes are less than 0.5%. From 16 nodes upwards we observe differences up to 6.2% at 256 nodes, which are compatible with empirical observations of running experiments on a non-dedicated system.

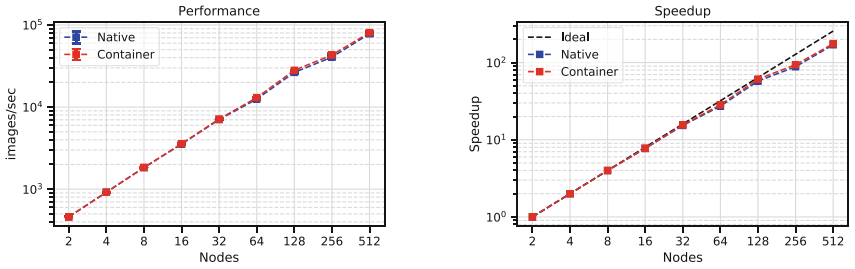

Fig. 3. Performance and speedup comparison between native and container versions of TensorFlow with Horovod.

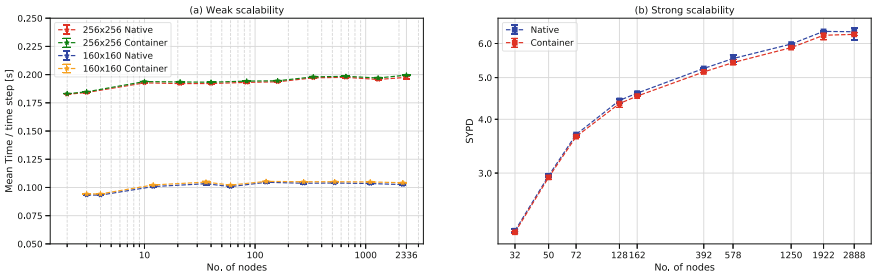

Fig. 4. Weak (a) and strong (b) scalability comparison between native and container versions of the COSMO code.

COSMO Atmospheric Model Code. The Consortium for Small Scale Modeling (COSMO) [6] develops and maintains a non-hydrostatic, limited-area atmospheric model which is used by several institutions for climate simulations [4,21] and production-level numerical weather prediction (NWP) [3,20,34] on a daily basis.

As test cases, we replicate the strong and weak scaling experiments performed in [13]. These experiments use a refactored version 5.0 of the GPU-accelerated COSMO code to simulate an idealized baroclinic wave [15]. The setup expands the computational domain of COSMO to cover 98.4% of the Earth surface and discretizes the vertical dimension using 60 stretched mode levels, going from the surface up to 40 km. For the complete details of the experimental setup, please refer to [13].

For the native application, we build the COSMO code and its dependencies on the Cray Programming Environment 18.08 available on Piz Daint. For the container application, in order to meet the specific requirements of the COSMO code, we package the Cray Linux Environment 6.0UP07 and the Cray Developer Toolkit 18.08 in a Docker image. This image is subsequently used as the base environment in which the containerized COSMO stack is built.

Weak scaling. The experiments for weak scaling employ two domain sizes with 160×160 and 256×256 horizontal grid points per node, respectively. We

perform runs from a minimum of 2 nodes up to 2336, carrying out 5 repetitions at each node count. For each run, the mean wall-clock time of a simulation step is calculated by taking the total duration of the time loop as reported by the application log and dividing it for the number of time steps. Figure 4(a) displays the average, minimum and maximum values of mean step duration for each data point. The performance trend of native and container applications is identical across all node counts and domain sizes. The absolute performance differences range from 0.3% to 1% on the 256×256 per-node domain and from 1% to 1.6% on the 160×160 per-node domain, which are within the observed variability for running on a shared system.

Strong scaling. The experiments for strong scaling employ a fixed domain with a 19 km horizontal grid spacing. We perform runs from a minimum of 32 nodes up to 2888, carrying out 5 repetitions at each node count. We measure performance in simulated years per day (SYPD), calculated from the simulation time step and the wall-clock duration of the time loop as reported by the application log. Figure 4 (b) displays the average, minimum and maximum values of SYPD for each data point. Again, native and container performances follow very similar trends throughout all deployment scales and show very small variations. Differences range from 0.8% to 2.1%. In the same fashion of the other experiments, such differences are consistent with previous experience about not having the system dedicated to these experiments.

6 Conclusions

Sarus is a container engine for HPC enabling the deployment of Docker containers which achieve native performance and scalability on supercomputing platforms.

Being designed around the specifications of the OCI, Sarus uses external hooks with a standard interface in order to extend its capabilities and support unique features of HPC environments, like system-tuned libraries and vendor-optimized drivers for dedicated hardware (e.g. accelerators and high-performance interconnects). Sarus integrates an OCI-compliant runtime, such as `runc`, to leverage low-level kernel features while reducing development efforts and architectural complexity. Sarus complements the advantages of the OCI specifications by tailoring containers for multi-tenant cluster systems, with regard to security, access permissions, and superior I/O performance for container images mounted on network parallel filesystems. The capability of importing Docker images and a Docker-compatible CLI increase the convenience of packaging software on personal computing devices, and then using Sarus to deploy containers at scale on HPC systems.

We showed that containerized applications deployed with Sarus on the Piz Daint supercomputing system can perform on par with native implementations by testing three real-world applications with complex dependency stacks: the GROMACS package for molecular dynamics, the combination of TensorFlow and

Horovod frameworks for Machine Learning, and the COSMO atmospheric model simulation code. The presented results show no performance degradation when comparing the natively compiled versions with their containerized counterparts, even when deploying containers on 2888 GPU nodes.

References

1. Abadi, M., et al.: TensorFlow: a system for large-scale machine learning. In: Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI), Savannah, Georgia, USA (2016)
2. Abraham, M.J., et al.: GROMACS: high performance molecular simulations through multi-level parallelism from laptops to supercomputers. SoftwareX **1**, 19–25 (2015)
3. Baldauf, M., Seifert, A., Förstner, J., Majewski, D., Raschendorfer, M., Reinhardt, T.: Operational convective-scale numerical weather prediction with the cosmo model: description and sensitivities. Mon. Weather Rev. **139**(12), 3887–3905 (2011)
4. Ban, N., Schmidli, J., Schär, C.: Heavy precipitation in a changing climate: does short-term summer precipitation increase faster? Geophys. Res. Lett. **42**(4), 1165–1172 (2015)
5. Brown, N.: Overlay Filesystem (2014). https://www.kernel.org/doc/Documentation/filesystems/overlayfs.txt
6. Consortium for Small-Scale Modeling: Cosmo model (2019). http://www.cosmo-model.org/
7. CRAY: XC Series System Administration Guide (S-2393) Rev B, February 2019. https://pubs.cray.com/content/S-2393/CLE%206.0.UP06/xctm-series-system-administration-guide/about-xctm-series-system-administration-guide-s-2393
8. Deal, S.J.: HPC made easy: using docker to distribute and test trilinos (2016)
9. Delaruelle, X.: Environment Modules open source project. http://modules.sourceforge.net/
10. Essmann, U., Perera, L., Berkowitz, M.L., Darden, T., Lee, H., Pedersen, L.G.: A smooth particle mesh ewald method. J. Chem. Phys. **103**(19), 8577–8593 (1995)
11. MPI Forum: MPI: a message-passing interface standard. version 3.0, September 2012. http://www.mpi-forum.org
12. OpenSSH Foundation: OpenSSH (1999). https://www.openssh.com/
13. Fuhrer, O., et al.: Near-global climate simulation at 1 km resolution: establishing a performance baseline on 4888 GPUs with COSMO 5.0. Geosci. Model Dev. 11(4), 1665–1681 (2018)
14. He, K., Zhang, X., Ren, S., Sun, J.: Deep residual learning for image recognition. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pp. 770–778 (2016)
15. Jablonowski, C., Williamson, D.L.: A baroclinic instability test case for atmospheric model dynamical cores. Q. J. R. Meteorol. Soc. **132**(621C), 2943–2975 (2006)
16. Jacobsen, D.M., Canon, R.S.: Contain this, unleashing Docker for HPC. In: Proceedings of the Cray User Group (2015)
17. Joubert, W., et al.: Attacking the opioid epidemic: determining the epistatic and pleiotropic genetic architectures for chronic pain and opioid addiction. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis, SC 2018, Piscataway, NJ, USA, pp. 57:1–57:14. IEEE Press (2018). http://dl.acm.org/citation.cfm?id=3291656.3291732

18. Kurth, T., et al.: Exascale deep learning for climate analytics. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis, SC 2018, Piscataway, NJ, USA, pp. 51:1–51:12. IEEE Press (2018). http://dl.acm.org/citation.cfm?id=3291656.3291724
19. Kurtzer, G.M.: Singularity 2.1.2 - Linux application and environment containers for science, August 2016. https://doi.org/10.5281/zenodo.60736
20. Lapillonne, X., et al.: Operational numerical weather prediction on a GPU-accelerated cluster supercomputer. In: EGU General Assembly Conference Abstracts, vol. 18 (2016)
21. Leutwyler, D., Lüthi, D., Ban, N., Fuhrer, O., Schär, C.: Evaluation of the convection-resolving climate modeling approach on continental scales. J. Geophys. Res.: Atmos. **122**(10), 5237–5258 (2017)
22. Hykes, S.: Spinning Out Docker's Plumbing: Part 1: Introducing runC (2015). https://blog.docker.com/2015/06/runc/. Accessed 22 November 2018
23. Linux man-pages project: ldconfig(8), September 2017. http://man7.org/linux/man-pages/man8/ldconfig.8.html
24. Linux man-pages project: capabilities(7), February 2018. http://man7.org/linux/man-pages/man7/capabilities.7.html
25. Linux man-pages project: mount_namespaces(7), April 2018. http://man7.org/linux/man-pages/man7/mount_namespaces.7.html
26. Linux man-pages project: unshare(2), February 2018. http://man7.org/linux/man-pages/man2/unshare.2.html
27. MPICH: MPICH ABI Compatibility Initiative, November 2013. https://www.mpich.org/abi/
28. NVIDIA: NVIDIA Container Runtime (2018). https://developer.nvidia.com/nvidia-container-runtime
29. Open Container Initiative: The 5 principles of Standard Containers (2015). https://github.com/opencontainers/runtime-spec/blob/master/principles.md
30. Open Containers Initiative: runC: CLI tool for spawning and running containers according to the OCI specification (2014), https://github.com/opencontainers/runc
31. OpenSFS and EOFS: Lustre filesystem. http://lustre.org
32. PRACE: Unified European Applications Benchmark Suite, October 2016. http://www.prace-ri.eu/ueabs/
33. 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, p. 36. ACM (2017)
34. Richard, E., Buzzi, A., Zängl, G.: Quantitative precipitation forecasting in the Alps: the advances achieved by the mesoscale alpine programme. Q. J. R. Meteorol. Soc. **133**(625), 831–846 (2007)
35. SchedMD: Slurm Workload Manager, November 2018. https://slurm.schedmd.com
36. Sergeev, A., Del Balso, M.: Horovod: fast and easy distributed deep learning in TensorFlow. arXiv preprint arXiv:1802.05799 (2018)
37. Shimokawabe, T., et al.: Peta-scale phase-field simulation for dendritic solidification on the TSUBAME 2.0 supercomputer. In: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis, p. 3. ACM (2011)
38. TensorFlow Project: Tensorflow Benchmarks (2015). https://github.com/tensorflow/benchmarks
39. TOP500.org: November 2018—TOP500 Supercomputer sites (2018). https://www.top500.org/lists/2018/11/

40. Vincent, P., Witherden, F., Vermeire, B., Park, J.S., Iyer, A.: Towards green aviation with Python at petascale. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, p. 1. IEEE Press (2016)
41. Ylonen, T., Lonvick, C.: The Secure Shell (SSH) protocol architecture (2005)
42. Zaharia, M., et al.: Apache spark: a unified engine for big data processing. Commun. ACM **59**(11), 56–65, October 2016. https://doi.org/10.1145/2934664
43. Zak, K.: mount(8), August 2015. http://man7.org/linux/man-pages/man8/mount.8.html