Containerization for High Performance Computing Systems: Survey and Prospects

Naweiluo Zhou, Huan Zhou, and Dennis Hoppe

***Abstract*—Containers improve the efficiency in application deployment and thus have been widely utilised on Cloud and lately in High Performance Computing (HPC) environments. Containers encapsulate complex programs with their dependencies in isolated environments making applications more compatible and portable. Often HPC systems have higher security levels compared to Cloud systems, which restrict users' ability to customise environments. Therefore, containers on HPC need to include a heavy package of libraries making their size relatively large. These libraries usually are specifically optimised for the hardware, which compromises portability of containers. *Per contra*, a Cloud container has smaller volume and is more portable. Furthermore, containers would benefit from orchestrators that facilitate deployment and management of containers at a large scale. Cloud systems in practice usually incorporate sophisticated container orchestration mechanisms as opposed to HPC systems. Nevertheless, some solutions to enable container orchestration on HPC systems have been proposed in state of the art. This paper gives a survey and taxonomy of efforts in both containerisation and its orchestration strategies on HPC systems. It highlights differences thereof between Cloud and HPC. Lastly, challenges are discussed and the potentials for research and engineering are envisioned.**

***Index Terms*—AI, cloud computing, container, HPC, job scheduling, orchestration, resource management.**

I. Introduction

CONTAINERS have been widely adopted on Cloud systems. Applications together with their dependencies are encapsulated into containers [1], which can ensure environment compatibility and enable users to move and deploy programs easily among clusters. Containerisation is a virtualisation technology [2]. Rather than creating an entire operating system (called guest OS) on top of a host OS as in a Virtual Machine (VM), containers only share the host kernel, which makes containers more lightweight than VMs. Containers on Cloud are often dedicated to run *micro-services* [3] and one container mostly hosts one application or a part of it.

High Performance Computing (HPC) systems are traditionally employed to perform large-scale financial, engineering and scientific simulations [4] that demand low latency (e.g., interconnect) and high throughput (e.g., the number of jobs completed over a specific time). To satisfy different user requirements, HPC systems normally provide predefined modules with specific software versions that users can switch by loading or unloading the modules with the desired packages [5]. This approach requires assistance of system administrators and therefore limits increasing user demands for environment customisation. On a multi-tenant environment as on HPC systems, especially HPC production systems, installation of new software packages on-demand by users is restricted, as it may alter the working environments of existing users and even raise security risks. Module-enabled software environments are also inconvenient for dynamic Artificial Intelligence (AI) software stacks [6]. Big Data Analytics hosted on Cloud are compute-intensive or data-intensive, mainly due to deployments of AI or Machine Learning (ML) applications, which demand extremely fast knowledge extraction in order to make rapid and accurate decisions. HPC-enabled AI can offer optimisation of supply chains, complex logics, manufacturing, simulation and underpin modelling to solve complex problems [7]. Typically, AI applications have sophisticated requirements of software stacks and configurations. Containerisation not only enables customised environments on HPC systems, but also brings research reproducibility into practice.

Containerised applications can become complex, e.g., thousands of separate containers may be required in production, and containers may require network isolation among each other for security reasons. Sophisticated strategies for container orchestration [8] have been developed on Cloud or big-data clusters to meet such requirements. HPC systems, *per contra*, lack features of efficiency in container scheduling and management (*e.g.* load balancing and auto container scaling), and often provide no integrated support for environment provisioning (i.e., infrastructure, configurations and dependencies).

There have been numerous studies on containerisation and container orchestration on Cloud [2], [9], [10], [11], [12], [13], [14], [15], however, there is no comprehensive survey on these technologies and techniques for HPC systems existing as of yet. This article:

- Investigates state-of-the-art works in containerisation on HPC systems and underscores their differences with respect to the Cloud;
- Introduces the representative orchestration frameworks on both HPC and Cloud environments, and highlights their feature differences;

Manuscript received 25 May 2022; revised 8 October 2022; accepted 21 November 2022. Date of publication 14 December 2022; date of current version 18 April 2023. This work was supported in part by the European Union's Horizon 2020 Research and Innovation Programme under Grant 825355 (CYBELE), and in part by the Ministry of Science, Research and the Arts of the State of Baden-Württemberg, Germany, through the Project CATALYST. Recommended for acceptance by R. Kazman. *(Corresponding author: Naweiluo Zhou.)*

The authors are with the High Performance Computing Center Stuttgart (HLRS), University of Stuttgart, 70569 Stuttgart, Germany (e-mail: naweiluo.zhou@ieee.org; huan.zhou@hlrs.de; hoppe@hlrs.de).

Digital Object Identifier 10.1109/TSE.2022.3229221

- Gathers the related studies in the integration of container orchestration strategies on Cloud into HPC environments;
- Discusses the challenges and envisions the potential directions for research and engineering.

The rest of the paper is organised as follows. First, Section II introduces the background on containerisation technologies and techniques. Key technologies of state-of-the-art container engines (Section III) and orchestration strategies (Section IV) are presented, and the feature differences thereof between HPC and Cloud systems are discussed. Next, Section V describes research challenges and the vision. Lastly, Section VI concludes this paper.

II. Concepts and Technologies for Containerisation

The main differences between containerisation technologies on Cloud and HPC systems are in terms of security and the types of workloads. The HPC applications tend to require more resources as to not only CPUs, but also the amount of memory and network speed. HPC communities have, therefore, developed sophisticated workload managers to leverage hardware resources and optimise application scheduling. Since the typical applications on Cloud differ significantly from those in HPC centres with respect to the sizes, execution time and requirements of the availability of hardware resources [16], the management systems on Cloud are evolved to include architectures different from those on HPC systems.

Research and engineering on containerisation technologies and techniques for HPC systems can be classified into two broad categories:

1) Container engines/runtimes;
2) Container orchestration.

In the first category, various architectures of container engines have been developed which vary in usage of namespaces (see Section II-A), image formats and programming languages. The research in the latter category is still in its primitive stage, which will be discussed in Section IV.

A. Containerisation Concepts

Containerisation is an OS-level virtualisation technology [17] that provides separation of application execution environments. A container is a runnable instance of an image that encapsulates a program together with its libraries, data, configuration files, *etc.* [1] in an isolated environment, hence it can ensure library compatibility and enables users to move and deploy programs easily among clusters. A container utilises the dependencies in its host kernel. The host merely needs to start a new process that is isolated from the host itself to boot a new container [18], thus making container start-up time comparable to that of a native application. In contrary, a traditional VM loads an entire guest kernel (simulated OS) into memory, which can occupy gigabytes of storage space on the host and requires a significant fraction of system resources to run. VMs are managed by *hypervisor* which is also known as Virtual Machine Monitor (VMM) that partitions and provisions VMs with hardware resources (e.g., CPU and memory). The hypervisor gives the hardware-level virtualisation [19], [20]. Fig. 1 highlights the architecture distinction of VMs and containers. It is worth noting that containers can also run inside VMs [21]. Besides portability, containers also enable reproducibility, *i.e.* once a program has been defined inside the container, its included working environment remains unchanged regardless of its running occurrences. Nevertheless, the shared kernel strategy presents an obvious pitfall: a Windows containerised application cannot execute on Unix kernels. Obviously, this should not become an impediment to its usage as Unix-like OS are often the preference for HPC systems.

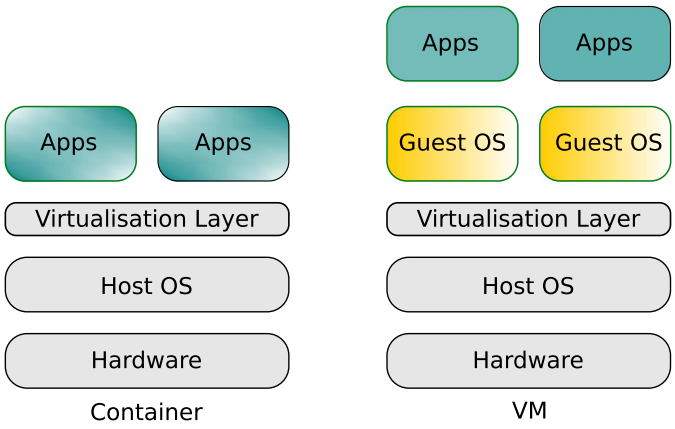

Fig. 1. Structure comparison of VMs and containers. On the VM side, the virtualisation layer often appears to be hypervisor while on the container side it is the container runtimes.

HPC applications are often highly optimised for processor architectures, interconnects, accelerators and other hardware aspects. Containerised applications, therefore, need to compromise between performance and portability. The studies have shown that containers can often achieve near-native performance [18], [22], [23], [24], [25], [26], [27] (see Section III-B).

Linux has several namespaces [28] that isolate various kernel resources: `mount` (file system tree and mounts), `PID` (process ID), `UTS` (hostname and domain name), `network` (e.g., network devices, ports, routing tables and firewall rules), `IPC` (inter-process communication resources) and `user`. The last namespace is an unprivileged namespace that grants the unprivileged process access to traditional privileged functionalities under a safe context. More specifically, the `user` namespace allows to map user ID (UID) and group ID (GID) from hosts to containers, meaning that a user having UID 0 (root) inside a container can be mapped to a non-root ID (e.g., 100000) outside the container. `Cgroups` (Control Groups) is another namespace that is targeted to limit, isolate and measure resource consumption of processes. `Cgroups` is useful for a multi-tenant setting as excess resource consumption of certain users will be only adverse to themselves. One application of Linux namespaces is the implementation of containers, e.g., Docker, the most widely-used container engine, uses namespaces to provide the isolated workspace that is called *container*. When a container executes, Docker creates a set of namespaces for that container.

B. Docker

There are multiple techniques that realise the concept of containers. Docker is among the most popular ones [27]. After its appearance in 2013, various container solutions aimed for HPC

TABLE I
LINUX NAMESPACE SUPPORTS FOR HPC-TARGETED CONTAINER ENGINES (SECTION III) AND DOCKER IN THE YEAR OF 2022

Namespaces	Singularity	Shifter	Charliecloud	UDocker	SARUS	Docker
mount	✓	✓	✓	✓	✓	✓
PID	✓	✗	✗	✗	✗	✓
UTS	✗	✗	✗	✗	✗	✓
network	✗	✗	✗	✗	✗	✓
IPC	✗	✗	✗	✗	✗	✓
user	✓	✓	✓	✓	✓	✓
Cgroup	✓	✓	✗	✗	✓	✓

Note that without certain namespaces, containers may still operate however with restricted functionalities.

have emerged [22]. Docker, initially based on LXC [29], is a container engine that supports multiple platforms, i.e., Linux, OSX and Windows. A Docker container image is composed of a readable/writable layer above a series of read-only layers. A new writable layer is added to the underlying layers when a new Docker container is created. All changes that are made to the running container, such as writing new files, modifying or deleting existing files, are written to this thin writable container layer. Docker adopts namespaces including `Cgroups` to provide resource isolation and resource limitation, respectively. Table I highlights the usage of namespaces with respect to Docker and a list of container engines targeted for HPC environments.

Docker provides network isolation and communication by creating three types of networks: *host*, *bridge* and *none*. The bridge network is the default Docker network. The Docker engine creates a subset or gateway to the bridged network. This software *bridge* allows Docker containers to communicate within the same bridged network; meanwhile, isolates the containers from a different bridged network. Containers in the same host can communicate via the default network by the host IP address. To communicate with the containers located on a different host, the host needs to allocate ports on its IP address. Managing ports brings overhead which can intensify at scale. Dynamically managing ports can solve this issue which is better handled by orchestration platforms as introduced in Section IV-B.

Docker is widely adopted in Cloud where users often have root privileges. The root privilege is required to execute the Docker application and its Daemon process that provides the essential services. Originally running Docker with root permission brings some advantages to Cloud users. For instance, users can run their applications and alternative security modules to provide separation among different allocations [30]; users can also mount host filesystems to their containers. Root privilege can cause security issues. Therefore, the latest updates of Docker engine start to support rootless daemon and enable users to execute containers without root. Nevertheless, other security concerns still persist. For instance, usage of Unix socket can be changed to TCP socket which will grant an attacker a remote control to execute any containers in the privileged mode. Additionally, rootless Docker does not run out of box, system administrators need to carefully set the namespaces of hosts to separate resources and user groups in order to guarantee security. Hence HPC centres that typically have high security requirements are still reluctant to enable the Docker support on their systems.

III. CONTAINER ENGINES AND RUNTIMES FOR HPC SYSTEMS

This section first reviews the state-of-the-art container engines/runtimes designed for HPC systems and compares the major differences with the mainstream Cloud container engine, i.e., Docker. Next, Section III-B shows the performance evaluation of the reviewed HPC container engines.

A. State-of-The-Art Container Engines and Runtimes

A list of representative container engines and runtimes for HPC systems is given in this section. They differ in functional extent and implementation, however, also hold some similarities. Tables I and II summarise the feature differences and similarities between Docker and a list of main HPC container engines.

1) Shifter: Shifter [31] is a prototypical implementation of container engine for HPC developed by NERSC. It utilises Docker for image building workflow. Once an image is built, users can submit it to an unprivileged gateway which injects configurations and binaries, flattens it to an ext4 file system image, and then compresses to squashfs images that are copied to a parallel filesystem on the nodes. In this way, Shifter insulates the network filesystem from image metadata traffic. Root permission of Docker is naturally deprived from Shifter that only grant user-level privileges. Existing directories can be also mounted inside Shifter image by passing certain flags.

As an HPC container engine, Shifter supports MPICH that is an implementation of the Message Passing Interface (MPI) [32], [33] standard. To enable accelerator supports such as GPU without compromising container portability, Shifter runtime swaps the built-in GPU driver of a Shifter container with an ABI (Application Binary Interface) compatible version at the container start-up time.

2) Charliecloud: Charliecloud [28] runs containers without privileged operations or daemons. Charlicloud can convert a Docker image into a tar file and unpacks it on the HPC nodes. Installation of Charliecloud does not require root permission. Such non-intrusive mechanisms are ideal for HPC systems. Charliecloud is considered to be secure against *shenanigans*, such as `chroot` escape, bypass of file and directory permission, privileged ports bound to all host IP addresses or UID set to an unmapped UID [15].

TABLE II
COMPARISON OF DOCKER WITH THE LIST OF CONTAINER ENGINES FOR HPC SYSTEMS

Container engines	Docker	Singularity	Shifter	Charliecloud	SARUS	UDocker
Usage of namespaces	✓	✓	✓	✓	✓	✓
MPI support	✓	✓	✓	✓	✓	✓
GPU support	✓	✓	✓	✓	✓	✓
Network support	Pluggable network driver (*e.g.* bridge)	Host network	Host network	Host network	Host network	Host network
Image format	Layers of files	Single image file, Filesystem bundle	squashfs layers	Filesystem bundle	Filesystem bundle	layers of files
Access to host filesystems	✓	✓	✓	✓	✓	✓
Escalation of permission	✓	✗	✗	✗	✗	✗
Privileged daemon*	✓	✗	✗	✗	✗	✗
Orchestration	Docker Swarm	HPC WLM	HPC WLM	HPC WLM	HPC WLM	HPC WLM
Programming languages	Go	Go	C	C	C++	Python

**Starting from v19.03, Docker also provides options to change its daemon to be rootless.*
WLM: workload manager. Orchestration is described in section iv-a and section iv-b.

MPI is supported by Charliecloud. Injecting host files into images is used by Charliecloud to solve library compatibility issues, such as GPU libraries that may be tied to specific kernel versions.

3) Singularity: Singularity is the most-widely used HPC container engine in academia and industry. Singularity [34] was specifically designed from the outset for HPC systems. Contrasting with Docker, it gives the following merits [23]:

- Running with user privileges and no daemon process. Only user privileges are required to execute Singularity applications. Acquisition of root permission is only necessary when users want to build or rebuild images, which can be performed on their own working computers. Unprivileged users can also build an image from a definition file with a few restrictions by ”fake root” in Singularity, however, some methods requiring to create block devices (e.g., `/dev/null`) may not always work correctly in this way;
- Seamless integration with HPC systems. Singularity natively supports GPU, MPI and InfiniBand [16]. No additional network configurations are expected in contrast with Docker containers;
- Portable via a single image file (SIF format). On the contrary, Docker is built up on top of layers of files.

Two approaches are often used to execute MPI applications using Singularity, i.e., hybrid model and bind model. The former compiles MPI binaries, libraries and the MPI application into a Singularity image. The latter binds the container on a host location where the container utilises the MPI libraries and binaries on the host. The latter model has a smaller image size since it does not include compiled MPI libraries and binaries in the image. Utilising the host libraries is also beneficial to application performance, however, the version of MPI implementation that is used to compile the application inside the container must be compatible with the version available on the host. The hybrid model is recommended, as mounting storage volumes on the host often require privileged operations.

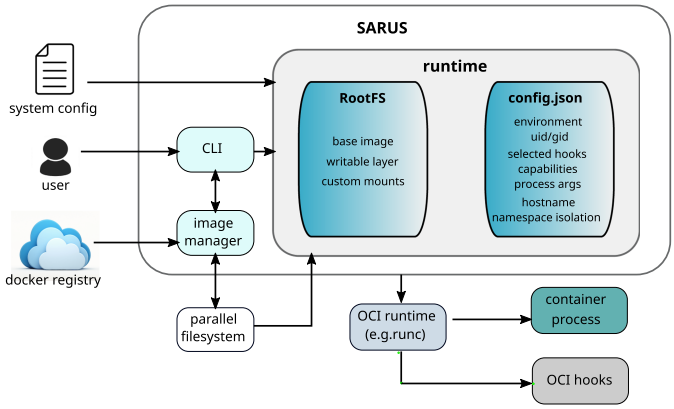

Fig. 2. The internal structure of SARUS. OCI hooks include MPI hook.

Most Docker images can be converted to singularity images directly via simple command lines (*e.g.* `docker save`, `singularity build`). Singularity has quickly become the *ipso facto* standard container engine for HPC systems.

4) SARUS: SARUS [35] is another container engine targeted for HPC systems. SARUS relies on `runc`[1] to instantiate containers. `runc` is a CLI (Command-Line Interface) tool for spawning and running containers according to the OCI (Open Container Initiative) specification. Different from the aforementioned engines, the internal structure of SARUS is based on the OCI standard (see Section V-A2). As shown in Fig. 2, the CLI component takes the command lines which either invoke the *image manager* component or the *runtime* component. The latter instantiates and executes containers by creating a bundle that comprises a root filesystem directory and a JSON configuration file. The *runtime* component then calls `runc` that will spawn the container processes. It is worth noting that functionalities of

[1][Online]. Available: https://github.com/opencontainers/runc

TABLE III
OVERVIEW OF THE RELATED WORK ON CONTAINER PERFORMANCE EVALUATION IN TERMS OF CPU, MEMORY, DISK, NETWORK AND GPU ON HPC SYSTEMS

Metrics	Performance Overhead	Work	Container engines	HPC vendors
CPU time	Often negligible. Large overhead caused by vendor-tuned libraries and dynamically linking libraries; better performance in many-process Python programs	[18], [22], [23], [24], [27], [40]	Singularity, Shifter, Charliecloud, Podman, SARUS	Cray XC, Cray XE/XK
Memory usage	Negligible	[23], [27]	Singularity, Charliecloud, Podman	-
Disk usage	Negligible	[27]	Singularity, Charliecloud, Podman	-
Network	Negligible or slight overhead. Overhead happens at start-up time because of the single file/bundle structure	[23], [24], [27]	Singularity, Charliecloud, Podman	Cray XC
GPU	A slight overhead	[23], [41]	Singularity	IBM

SARUS can be extended by calling customised OCI hooks, e.g., MPI hook.

5) UDocker: UDocker[2] is a Python wrapper for the Docker container, which executes only simple Docker containers in user space without the acquisition of root privileges. UDocker provides a Docker-like CLI and only supports a subset of Docker commands, i.e., `search`, `pull`, `import`, `export`, `load`, `save`, `create` and `run`. It is worth noting that UDocker neither makes use of Docker nor requires its presence on the host. It executes containers by simply providing a chroot-like environment over the extracted container.

6) Other HPC Container Engines: More and more HPC container engines are being developed, this section gives an overview of some that are targeted for special use cases.

Podman [36] makes use of the `user` namespace to execute containers without privilege escalation. A Podman container image comprises layers of read-write files as Docker. It adopts the same runtime `runc` as in SARUS and Docker. The runtime `crun`, which is faster than `runc`, is also supported. A notable feature of Podman is as its name denotes: the concept of *pod*. A pod groups a set of containers that collectively implements a complex application to share namespaces and simplify communication. This feature enables the convergence with the Kubernetes [37] environment (Section IV-B1), however, requires advanced kernel features (e.g., version 2 `Cgroups` and user-space FUSE). These kernel features are not yet compatible with network filesystems to make full use of the rootless capabilities of Podman and consequently restrain its usage from HPC production systems [38].

Similar to UDocker, Socker [39] is a simple secure wrapper to run Docker in HPC environments, more specifically SLURM (Section IV-A2). It does not support the `user` namespace, however, it takes the resource limits imposed by SLURM.

Enroot[3] from NVIDIA can be considered as an enhanced unprivileged `chroot`. It removes much of the isolation that the other container engines normally provide but preserves filesystem separation. Enroot makes use of `user` and `mount` namespaces.

TABLE IV
THE LIST OF HPC BENCHMARKS MENTIONED IN SECTION III-B

Benchmarks	Description
IMB	Intel MPI Benchmark
HPCG	A complement to the LINPACK
Linpack	Measure floating-point computing power
NAMD	Simulation for molecular dynamics
VASP	Atomic scale materials modelling
WRF	Weather Research and Forecasting Model
AMBER	Assisted Model Building with Energy Refinement
HPGMG-FE	High-Performance Geometric Multigrid

B. Performance Evaluation for HPC Container Engines

This section only selects the representative works as given in Table III, rather than exhausting the literature, to show the performance of containers that are specifically targeted for HPC systems in terms of CPU, memory, disk (I/O), network and GPU. Table VI lists the benchmarks utilised in these work. Overall, the container startup latency can be high on the Cloud. This startup overhead is caused by building containers from multiple image layers, setting read-write layers and monitoring containers [27]. An HPC container is composed of a single image or directory (with exception to Podman) and monitoring is performed by HPC systems.

The work in [24], utilising the IMB [42] benchmark suite and HPCG [43] benchmarks, proved that little overhead of network bandwidth and CPU computing overhead is caused by Singularity when dynamically linking vendor MPI libraries in order to efficiently leverage advanced hardware resources. With the Cray MPI library, Singularity container achieved 99.4% efficiency of native bandwidth on a Cray XC [44] HPC testbed when running the IMB benchmark suite. However, the efficacy drastically

[2][Online]. Available: https://github.com/indigo-dc/udocker
[3][Online]. Available: https://github.com/NVIDIA/enroot

drops to 39.5% with Intel MPI. Execution time evaluated with the HPCG benchmarks, indicated that the performance penalty caused by Singularity is negligible with Cray MPI, though the overhead can reach 18.1% with Intel MPI. The performance degradation with Intel MPI is mostly because of the vendor-tuned MPI library which does not leverage hardware resources from a different vendor, e.g., interconnect.

Hu et al. [23] evaluated the Singularity performance in terms of CPU capacity, memory, network bandwidth and GPU with Linpack benchmarks [45] and four typical HPC applications (i.e., NAMD [46], VASP [47], WRF [48] and AMBER [49]). Singularity provides close to native performance on CPU, memory and network bandwidth. A slight overhead (4.135%) is shown on NVIDIA GPU.

Muscianisi et al. [41] illustrated the performance impact of Singularity with the increasing number of GPU nodes. The evaluation was carried out on CINECA's GALILEO systems with TensorFlow [50] applications. The results again demonstrated that the container environments caused negligible performance overhead.

The work by Hale et al. [18] presented the CPU performance of Shifter with HPGMG-FE (MPI implementation) benchmarks [51] on Cray XC30 (192 cores, 24 cores per compute node) where the performance margin between Shifter container and bare metal is unnoticeable. Comparison is also given for MPI with implementation in C++ and Python using a custom benchmark. The authors observed that it could take over 30 minutes to import the Python modules when running natively with 1,000 processes. Each process of a Python application imports modules from the filesystem on each node. Accesses to many small files on an HPC filesystem using many processes can be extremely slow comparing with the accesses to a few large files. The containerised benchmark has already included all the modules in its image that is mounted as a single file on each node, therefore, Shifter container outperforms the native execution in this case. *Bahls* [40] also evaluated the execution time of Shifter on Cray XC and Cray XE/XK systems exploiting Cray HSN (High Speed Network). Their results showed that Shifter gave comparable performance to bare metal.

The study in [22] compared the performance of Shifter and Singularity against bare metal in terms of computation time using two biological use cases on three types of supercomputer CPU architectures: Intel Skylake, IBM Power9 and Arm-v8. Containerised applications can scale at the same rate as the bare-metal counterparts. However, the authors also observed that with a small number of MPI ranks, containers should be built as generic as possible, *per contra*, when it comes to a large number of cores, containers need to be tuned for the hosts.

Without performance comparison with bare-metal applications, the work in [27] studied the CPU, memory, network and I/O performance of Charliecloud, Podman and Singularity. All the containers behave similarly with respect to the CPU and memory usage. Charliecloud and Singularity have comparable I/O performance. Charliecloud incurs large overhead on Lustre's MDS (Metadata Server) and OSS (Object Storage Server) due to its bare tree structure. Comparing with the structures of shared layers (as in Docker), this structure needs to access a large number of individual files from the image tree from Lustre. Consequently, it causes network overhead when data is transmitted from the client node over the network at container start-up time. Similarly, as Singularity is stored as a single file on Lustre, a large amount of data needs to be loaded at starting point resulting in a data transmission spike on network.

SARUS has shown strong scaling capability on Cray XC systems with hybrid GPU and CPU nodes [35]. The performance difference between SARUS and bare metal is less than 0.5% up to 8 nodes and 6.2% up to 256 nodes. No specific metrics are given in terms of GPU, though GPU has been used as accelerators.

C. Section Highlights

Containers are introduced to HPC systems, as they enable environment customisation for users, which offers the solutions to application compatibility issues. This is particularly important on HPC systems that are typically inflexible for environment modifications. Notably, HPC container engines are designed to meet the high-security requirements on HPC systems. Multiple prevailing engines have been described in this section, they share some common features:

- Non-root privileges;
- Often can convert Docker images to their own image formats;
- Supports of MPI that are typical HPC applications;
- Use host network rather than pluggable network drivers.

Yet differences exist in their image formats. Layered image format is seen in Docker (UDocker wraps Docker image layers to a local directory), which is executed by pulling the image layers that have not been previously downloaded on the host. HPC container images are stored in a single directory or file which can be transferred to the compute nodes easily avoiding the pulling operations that require network access. HPC container engines show various ways to incorporate well-tuned libraries targeting for the hosts in order to achieve optimised performance, e.g., OCI hooks (SARUS), injecting host files into images (Charliecloud).

Section III-B aims to give examples that can provide general advices on how to build the container images to maximise performance. Clearly, performance loss can occur in certain cases which are summarised in the second column of Table III.

IV. Container Orchestration

Orchestration under the context herein means automated configuration, coordination and management of Cloud or HPC systems. In theory, HPC workload manager can be also addressed as orchestrator, however, this article takes the former term as it is the custom terminology that has been long-used and widely understood in the HPC area. The driving factors that push HPC workload managers and Cloud orchestrators to be developed in different directions can be multiple. This will be discussed at the end of this section (Section IV-D). However, first it is important to understand the mechanisms of HPC workload managers (Section IV-A) and Cloud orchestrators (Section IV-B). Mostly, container orchestration for HPC systems either relies on the orchestration strategies of the existing Cloud orchestrators or

TABLE V
COMPARISON OF HPC WORKLOAD MANAGERS (SECTION IV-A) AND CLOUD ORCHESTRATORS (SECTION IV-2)

	HPC workload manager	Cloud orchestrator
Deployment	Batch queue (queueing time from seconds to days)	Often immediate
Workload type	Binary	Container, pod
Supports of Parallel and Array Jobs	Both	Array*
Resource unit	Bare-metal nodes	Pods, VM nodes
Resource elasticity	No	Yes
Application execution length	Long duration & Run to completion	Continuously running or short duration†
Application specifics	Distributed memory jobs (*e.g.* MPI)	Often micro-services
DevOps environment provision	No	Yes
API supports	No (or very weak)	Yes
Job scheduling	Backfilling	On-demand scheduling
Centralised scheduling system	Yes	Not always
Job submission scripts	Batch scripts	Declarative files, typically `yaml` scripts
Checkpointing	Yes	No. Containers are relaunched upon failure
Support of multiple resource managers	No	Often yes

*Exceptions: *Mesos can support parallel jobs (Section 4.2.3); †YARN targets for long-running batch jobs (Section 4.2.3)*

exploits the mechanisms of current HPC workload managers or software tools. This point will be depicted in Section IV-C.

A. Workload Managers for HPC Systems

Cloud aims to exploit economy of scale by consolidating applications into the same hardware [16] and the hardware resources can be easily extended based on user demands. In contrast, HPC centres have large-scale hardware resources available and reserve computing resources exclusively for users. Table V underscores the main differences between HPC workload managers and Cloud orchestrators. A typical HPC system is managed by a workload manager. A *workload manager* comprises a *resource manager* and a *job scheduler*. A resource manager [52] allocates resources (e.g., CPU and memory), schedules jobs and guarantees no interference from other user processes. A job scheduler determines the job priorities, enforces resource limits and dispatches jobs to available nodes [53].

HPC workload managers incorporate a big family, such as PBS [54], Spectrum LSF [55], Grid Engine [56], OAR [57] and Slurm [58]. Slurm and PBS are two main-stream workload managers. The workload managers shares some common features: a centralised scheduling system, a queuing system and static resource management mechanisms, which will be detailed in this section.

1) PBS: PBS stands for Portable Batch System which includes three versions: OpenPBS, PBS Pro and TORQUE. OpenPBS is open-source and TORQUE is a fork of OpenPBS. PBS Pro is dual-licensed under an open-source and commercial license. The structure of a TORQUE-managed cluster consists of a head node and many compute nodes as illustrated in Fig. 3 where only three compute nodes are shown. The head node (coloured in blue in Fig. 3) controls the entire TORQUE system. A *pbs_server* daemon and a job scheduler daemon are located on the head node. The batch job is submitted to the head node

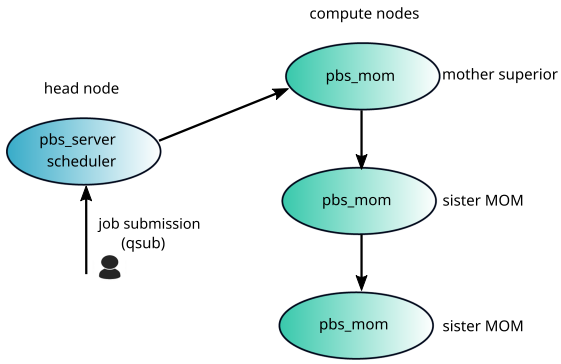

Fig. 3. TORQUE structure. *pbs_server*, scheduler and *pbs_mom* are the daemons running on the nodes. Mother superior is the first node on the node list (on step4).

(in some cases, the job is first submitted to a login node and then transferred to the head node). A node list that records the configured compute nodes is maintained on the head node. The architecture of this kind as shown in Fig 3 represents the fundamental cluster structure of main-stream HPC workload managers. The procedure of job submission on TORQUE is briefly described as follows:

1) The job is submitted to the head node by the command `qsub`. A job is normally written in the format of a PBS script. A job ID is returned to the user as the standard output of `qsub`.
2) The job record, which incorporates a job ID and the job attributes, is generated and passed to *pbs_server*.
3) *pbs_server* transfers the job record to the job scheduler daemon. The job scheduler daemon adds the job into a job queue and applies a scheduling algorithm to it (e.g., FIFO: First In First Out) which determines the job priority and its resource assignment.

4) When the scheduler finds the list of nodes for the job, it returns the job information to pbs_server. The first node on this list becomes the *mother superior* and the rest are called *sister MOMs* or *sister nodes*. pbs_server allocates the resources and passes the job control as well as execution information to the pbs_mom daemon installed on the mom superior node instructing to launch the job on the assigned compute nodes.
5) The pbs_mom daemons on the compute nodes manage the execution of jobs and monitor resource usage. pbs_mom will capture all the outputs and direct them to *stdout* and *stderr* which are written into the output and error files and are copied to the designated location when the job completes successfully. The job status (completed or terminated) will be passed to pbs_server by pbs_mom. The job information will be updated.

In TORQUE, nodes are partitioned into different groups called $queues$. In each queue, the administrator sets limits for resources such as walltime and job size. This feature can be useful for job scheduling in a large HPC cluster where nodes are heterogeneous or certain nodes are reserved for special users. This feature is commonly seen in HPC workload managers.

TORQUE has a default scheduler FIFO, and is often integrated with a more sophisticated job scheduler, such as Maui [59]. Maui is an open source job scheduler that provides advanced features such as dynamic job prioritisation, configurable parameters, extensive fair share capabilities and backfill scheduling. Maui functions in an iterative manner like most job schedulers. It starts a new iteration when one of the following conditions is met: (1) a job or resource state alters; (2) a reservation boundary event occurs; (3) an external command to resume scheduling is issued; (4) a configuration timer expires. In each iteration, Maui follows the below steps [60]:

1) Obtain resource records from TORQUE;
2) Fetch workload information from TORQUE;
3) Update statistics;
4) Refresh reservations;
5) Select jobs that are eligible for priority scheduling;
6) Prioritise eligible jobs;
7) Schedule jobs by priority and create reservations;
8) Backfill jobs.

Despite an abundance of algorithms, only a few scheduling strategies are practically in use by job schedulers. Backfilling scheduling [61] allows jobs to take the reserved job slots if this action does not delay the start of other jobs having reserved the resources, thus allowing large parallel jobs to execute and avoiding resource underutilisation. Differently, *Gang scheduling* [62] attempts to take care of the situations when the runtime of a job is unknown, allowing smaller jobs to get fairer access to the resources. Both scheduling strategies are also seen in SLURM and backfilling can be also found in LSF.

2) SLURM: The structure of a SLURM (Simple Linux Utility for Resource Management) [58] managed cluster is composed of one or two SLURM servers and many compute nodes. Its procedure of job submission is similar to that of TORQUE. Fig. 4 illustrates the structure of SLURM. Its server hosts the

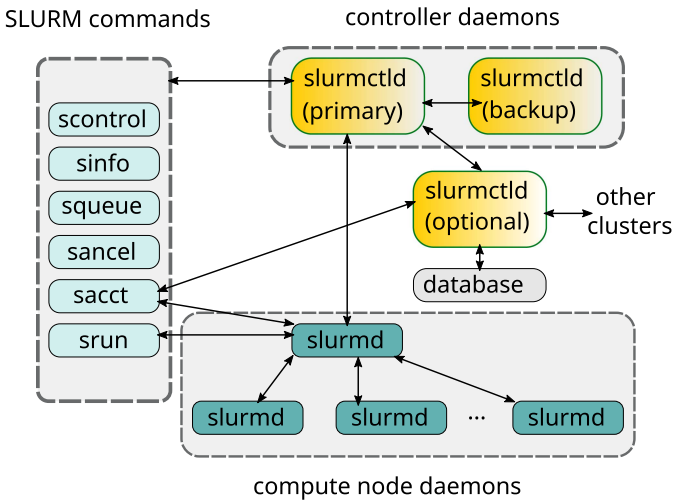

Fig. 4. SLURM structure.

`slurmctld` daemon which is responsible for cluster resource and job management. SLURM servers and the corresponding `slurmctld` daemons can be deployed in an active/passive mode in order to provide services of high reliability for computing clusters. Each compute node hosts one instance of the `slurmd` daemon, which is responsible for job staging and execution. There are additional daemons, e.g., `slurmdbd` which allows to collect and record accounting information for multiple SLURM-managed clusters and `slurmrestd` that can be used to interact with SLURM through a REST API (RESTful Application Programming Interface). The SLURM resource list is held as a part of the `slurm.conf` file located on SLURM server nodes, which contains a list of nodes including features (e.g., CPU speed and model, amount of memory) and configured *partitions* (named *queue* in PBS) including partition names, list of associated nodes and job priority.

Both PBS and SLURM have little (if at all) dedicated supports for container workloads. Containers are only scheduled as conventional HPC workloads, *e.g* lacking of load-balancing supports.

3) Spectrum LSF: IBM platform Load Sharing Facility (LSF), targeted for enterprises, is designed for distributed HPC deployments. LSF is based on the Utopia job scheduler [55] developed at the University of Toronto. Its Session Scheduler runs and manages short-duration batch jobs, which enables users to submit multiple tasks as a single LSF job, consequently reduces the number of job scheduling decisions. Session Scheduler can efficiently share resources regardless of job execution time and can make thousands of scheduling decisions per second. These capabilities create a focus on throughput which is often critical for HPC workloads. Fig. 5 illustrates the structure of LSF. Its license scheduler allows to make policies that control the way software licenses are shared among users within an organisation. Jobs are submitted via the command line interface, API or IBM platform application centre. Job submission carries similar procedure as in TORQUE.

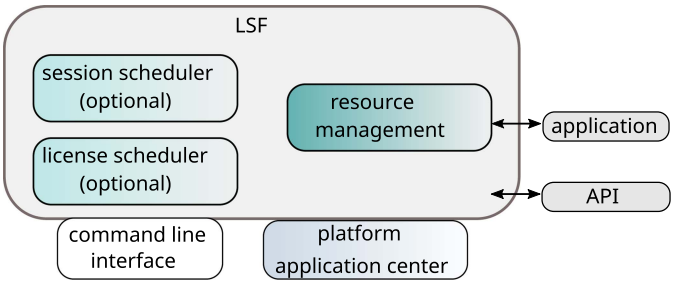

Fig. 5. Spectrum LSF structure.

LSF supports container workloads: Docker, Singularity and Shifter. LSF configures container runtime control in the *application profile*[4] that is managed by the system administrator. Users do not need to consider which containers are used for their jobs, instead only need to submit their jobs to the application profile and LSF automatically manages the container runtime control. Section IV-C3 elaborates this feature in more details.

B. Orchestration Frameworks on Cloud

Cloud clusters often include orchestration mechanisms to coordinate tasks and hardware resources. Cloud has evolved mature orchestrators to manage containers efficiently. Container orchestrators can offer [11], [15], [37]:

- Resource limit control. Reserve a specific amount of CPUs and memory for a container, which restrains interference from other containers and provides information for scheduling decisions;
- Scheduling. It determines the policies that optimise the placement of containers on nodes;
- Load balancing. It distributes workloads among container instances;
- Health check. It verifies if a faulty container needs to be destroyed or replaced;
- Fault tolerance. It allows to maintain a desired number of containers;
- Auto-scaling. It automatically adds and removes containers.

Additionally, a container orchestrator should also simplify networking, enable service discovery and support continuous deployment [63].

1) Kubernetes: Kubernetes originally developed by Google is among the most popular open-source container orchestrators, which has a rapidly growing community and ecosystem with numerous platforms being developed upon it. The architecture of Kubernetes comprises a master node and a set of worker nodes. Kubernetes runs containers inside *pods* that are scheduled to run either on master or worker nodes. A *pod* can include one or multiple containers. Kubernetes provides its services via *deployments* that are created by submission of *yaml* files. Inside a *yaml* file, users can specify services and computation to perform on the cluster. A user *deployment* can be performed either on the master node or the worker nodes.

Kubernetes is based on a highly modular architecture which abstracts the underlying infrastructure and allows internal customisation, such as the deployment of software-defined networks or storage solutions. It also supports various big-data frameworks, such as Hadoop MapReduce [64], Spark [65] and Kafka [66]. Kubernetes incorporates a powerful set of tools to control the life cycle of applications, e.g., parameterised redeployment in case of failures and state management. Furthermore, it supports software-defined infrastructures[5] [67] and resource disaggregation [68] by leveraging container-based deployments and particular *drivers* (e.g., Container Runtime Interface driver, Container Storage Interface driver and Container Network Interface driver) based on standardised interfaces. These interfaces enable the definition of abstractions for fine-grain control of computation, states and communication in multi-tenant Cloud environments along with optimal usage of the underlying hardware resources.

Kubernetes incorporates a scheduling system that permits users to specify different schedulers for each job. The scheduling system makes the decisions based on two steps before the actual scheduling operations:

1) Node filtering. The scheduler locates the node(s) that fit(s) the workload, e.g., a *pod* is specified with node affinity, therefore, only certain nodes can meet the affinity requirements or some nodes may not include enough CPU resources to serve the request. Normally the scheduler does not traverse the entire node list, instead it selects the one/ones detected first.
2) Node priority calculation. The scheduler calculates a score for each node, and the highest scoring node will run that *pod*.

Kubernetes has started being utilised to assist HPC systems in container orchestration (Section IV-C).

2) Docker Swarm: Docker Swarm [69] is built for the Docker engine. It is a much simpler orchestrator comparing with Kubernetes, e.g., it offers less rich functionalities, limited customisations and extensions. Docker Swarm is hence lightweight and suitable for small workloads. In contrast, Kubernetes is heavyweight for individual developers who may only want to set up an orchestrator for simplistic applications and perform infrequent deployments. Nevertheless, Docker Swarm still has its own API, and provides filtering, scheduling and load-balancing. API is a strong feature commonly used in Cloud orchestrators, as it enables applications or services to talk to each other and provides connections with other orchestrators.

The functionalities of Docker Swarm may be applied to perform container orchestration on HPC systems as detailed in Section IV-C3.

3) Apache Mesos and YARN: Apache Mesos [70] is a cluster manager that provides efficient resource isolation and sharing

[4]LSF application profile: it is used to refine queue-level settings, or to exclude some jobs from queue-level parameters.

[5]Software-defined infrastructure (SDI) is the definition of computing infrastructure entirely under the control of software with no operator or human intervention. It operates independent of any hardware-specific dependencies and is programmatically extensible.

across distributed applications or frameworks. Mesos removes the centralised scheduling model that would otherwise require to compute global schedules for all the tasks running on the different frameworks connected to Mesos. Instead, each framework on a Mesos cluster can define its own scheduling strategies. For instance, Mesos can be connected with MPI or Hadoop [71]. Mesos utilises a master process to manage slave daemons running on each node. A typical Mesos cluster includes $3 \sim 5$ masters with one acting as the leader and the rest on standby. The master controls scheduling across frameworks through *resource offers* that provide resource availability of the cluster to slaves. However, the master process only suggests the amount of resources that can be given to each framework according to the policies of organisations, *e.g* fair sharing. Each framework rules which resources or tasks to accept. Once a *resource offer* is accepted by a framework, the framework passes Mesos a description of the tasks. The slave comprises two components, i.e., a scheduler registered to the master to receive resources and an executor process to run tasks from the frameworks.

Mesos is a non-monolithic scheduler which acts as an arbiter that allocates resources across multiple schedulers, resolves conflicts, and ensures fair distribution of resources. Apache YARN (Yet Another Resource Negotiator) [72] is a monolithic scheduler which was developed in the first place to schedule Hadoop jobs. YARN is designed for long-running batch jobs and is unsuitable for long-running services and short-lived interactive queries.

Mesosphere Marathon[6] is a container orchestration framework for Apache Mesos. Literature has seen the usage of Mesos together with Marathon in container orchestration on HPC systems as detailed in Section IV-C3.

4) Ansible: Ansible [73] is a popular software orchestration tool. More specifically, it handles configuration management, application deployment, cloud provisioning, ad-hoc task execution, network automation and multi-node orchestration. The architecture of Ansible is simple and flexible, i.e., it does not require a special server or daemons running on the nodes. Configurations are set by *playbooks* that utilise *yaml* to describe the *automation jobs*, and connections to other nodes are via `ssh`. Nodes managed by Ansible are grouped into *inventories* that can be defined by users or drawn from different Cloud environments.

Ansible is adopted by the SODALITE framework (Section IV-C4) as a key component to automatically build container images.

5) OpenStack: OpenStack [74] is mostly deployed as infrastructure-as-a-service (IaaS)[7] [75] on Cloud. It can be utilised to deploy and manage cloud-based infrastructures that support various use cases, such as web hosting, Big Data projects, software as a service (SaaS) [76] delivery and deployment of containers, VMs or bare-metal. It presents a scalable and highly adaptive open source architecture for Cloud solutions and helps to leverage hardware resources [77]. It also manages heterogeneous compute, storage and network resources.

[6][Online]. Available: https://mesosphere.github.io/marathon/

[7]IaaS offers resources such as compute, storage and network as services to users based on demand.

Together with its support of containers, container orchestrators such as Docker Swarm, Kubernetes and Mesos, Openstack enables the possibilities to quickly deploy, maintain, and upgrade complex and highly available infrastructures. OpenStack is also used in HPC communities to provide IaaS to end-users, enabling them to dynamically create isolated HPC environments.

Academia and industry have developed a plethora of Cloud orchestrators. This article only reviews the ones that are mostly relevant to the HPC communities and the ones that have seen their usage in container orchestration for HPC systems, and the rest is out of the scope herein.

C. Bridge Orchestration Strategies Between HPC and Cloud

There are numerous works in literature [11], [78], [79], [80] on container orchestration for Cloud clusters, however, they are herein out of the scope. This section reviews the works that have been performed on the general issues of bridging the gap between conventional HPC and service-oriented infrastructures (Cloud). Overall, the state-of-the-art works on container orchestration for HPC systems fall into four categories as illustrated in Fig. 6.

1) Added functionalities to HPC workload managers. It relies on workload managers for resource management and scheduling; meanwhile adopts additional software such as MPI for container orchestration.
2) Connector between Cloud and HPC. Containers are scheduled from Cloud clusters to HPC clusters. This architecture isolates the HPC resources from Cloud so as to ensure HPC environment security; meanwhile offers application developments with flexible environments and powerful computing resources.
3) Cohabitation. Workload managers and Cloud orchestrators co-exist on an HPC cluster, such as IBM LSF-Kubernetes. This gives a direction for the provision of HPC resources as services. In practice, the HPC workload managers and Cloud orchestrators do not coexist in one cluster.
4) Meta-orchestration. An additional orchestrator is implemented on top of the Cloud orchestrator and HPC workload manager.

There are pros and cons of the above four categories, which are outlined in Table VI. In addition, a research and engineering trend [12], [30], [81], [82], [83] is to move HPC applications to Cloud, as Cloud provides flexible and cost-effective services which are favoured by small-sized or middle-sized business. Beltre et al. [84] proposed to manage HPC applications by Kubernetes on a Cloud cluster with powerful computing resources and InfiniBand, which demonstrated comparable performance in containerised and bare-metal environments. The approach of this kind may be extended to HPC systems, however, remains unpractical for HPC centres to completely substitute their existing workload managers.

1) Added Functionalities to WLM: A potential research direction is to complement workload managers with container orchestration or make use of the existing HPC software stacks.

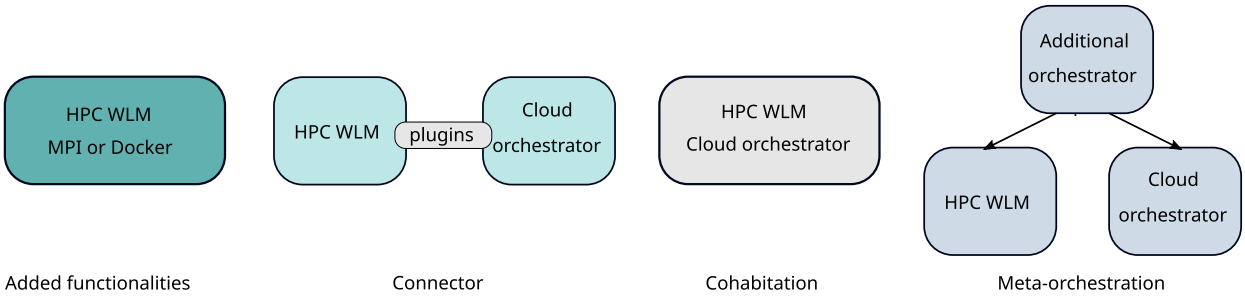

Fig. 6. The four types of container orchestration on HPC systems.

TABLE VI
A LIST OF THE RELATED WORK ON CONTAINER ORCHESTRATION FOR HPC SYSTEMS

Orchestration approaches	Advantages	Disadvantages
Added functionalities to WLM [85], [86], [87]	Less intrusive	Limited functionalities, security issues for usage of Docker on HPC
Connector between Cloud and HPC [88], [89], [90], [91]	Non-intrusive, flexible environments meanwhile power computing resources, exploit orchestration strategies of orchestration platforms	High network latency between Cloud and HPC
Cohabitation [1], [84], [86], [87], [92], [93], [94], [94]	Fully exploit the functionalities of orchestration platforms. flexible execution environments, enable HPC as services	Intrusive, security issues
Meta-orchestration [95], [96], [97]	Less-intrusive, flexible environments meanwhile power computing resources, container orchestration strategies in addition to the ones given by Cloud orchestrator	Increase architecture complexity, increase maintenance efforts

Wofford et al. [85] simply adopt Open Runtime Environment (orted) reference implementation from Open MPI to orchestrate container launch suitable for arbitrary batch schedulers.

Julian et al. [86] proposed their prototype for container orchestration in an HPC environment. A PBS-based HPC cluster can automatically scale up and down as load demands by launching Docker containers using the job scheduler Moab [98]. Three containers serve as the front-end system, scheduler (it runs PBS and Moab inside) and compute node (launches pbs_mom daemon, see Section IV-A1). More compute node containers are scheduled when there is no sufficient number of physical nodes. Unused containers are destroyed via external Python scripts when jobs complete. This approach may offer a solution for resource elasticity on HPC systems (Section V-B6). Similarly, an early study [87] described two models that can orchestrate Docker containers using an HPC workload manager. The former model launches a container to behave as one compute node which holds all assigned processes, whilst the latter boots a container per process by MPI launchers. The latter work seems to be outdated as to MPI applications which can be now automatically scaled with Singularity support.

2) Connector Between Cloud and HPC: Cloud technologies are evolving to be able to support complex applications of HPC, Big Data and AI. Nevertheless, the applications with intensive computation and high inter-processor communication could not scale well, particularly due to the lack of low latency networks (e.g., InfiniBand) and the usage of network virtualisation for network isolation. A research and development trend is to converge HPC and Cloud in order to take advantage of the resource management and scheduling of both HPC and Cloud infrastructures with minimal intrusion to HPC environments. Furthermore, the software stack and workflows in Cloud and HPC are usually developed and maintained by different organisations and users with various goals and methodologies, hence a connector between HPC and Cloud systems would bridge the gap and solve compatibility problems.

Zhou et al. [88], [89], [90], [91] described the design of a plugin named Torque-Operator that serves as the key component to its proposed hybrid architecture. The containerised AI applications are scheduled from the Kubernetes-managed Cloud cluster to the TORQUE-managed HPC cluster where the performance of the compute-intensive or data-demanding applications can be significantly enhanced. This approach is less intrusive to HPC systems, however, its architecture shows one drawback: the latency of the network bridging the Cloud and HPC clusters can be high, when a large amount of data needs to be transferred in-between.

DKube[8] is a commercial software that is able to execute a wide range of AI/ML components scheduled from Kubernetes to SLURM. The software comprises a Kubernetes plugin and a SLURM Plugin. The former is represented as a hub that

[8][Online]. Available: https://www.dkube.io/products/datascience/hpc-slurm.php

runs MLOps (Machine Learning Operations) management and associated Kubernetes workloads, while the latter connects to SLURM.

3) Cohabitation: Liu et al. [92] showed how to dynamically migrate computing resources between HPC and OpenStack clusters based on demands. At a higher level, IBM has demonstrated the ability to run Kubernetes *pods* on Spectrum LSF where LSF acts as a scheduler for Kubernetes. An additional Kubernetes scheduler daemon needs to be installed into the LSF cluster, which acts as a bridge between LSF and the Kuberentes server. `Kubelet` will execute and manage *pod* lifecycle on target nodes in the normal fashion. IBM released LSF connector to Kubernetes, which makes use of the core LSF scheduling technologies and Kubernetes API functionalities. Kubernetes needs to be installed in a subset of the LSF managed HPC cluster. This architecture allows users to run Kubernetes and HPC batch jobs on the same infrastructure. The LSF scheduler is packed into containers and users submit jobs via `kubectl`. The LSF scheduler listens to the Kubernetes API server and translates *pod* requests into jobs for the LSF scheduler. This approach can add additional heavy workloads to HPC systems, as Kubernetes relies deployments of services across clusters to perform load balancing, scheduling, auto scheduling, *etc*.

Piras et al. [93] implemented a method that expanded Kubernetes clusters with HPC clusters through Grid Engine. Submission is performed by PBS jobs to launch Kubernetes jobs. Therefore, HPC nodes are added to Kubernetes clusters by installing Kubernetes core components (i.e., `kubeadm` and `Kubelet`) and Docker container engine. On HPC, especially HPC production systems in HPC centres, adding new software packages that require using root privileges can cause security risks and alter the working environments of current users. The security issues will be further elaborated in Section V-A4.

Khan et al. [1] proposed to containerise HPC workloads and install Mesos and Marathon (Section IV-B3) on HPC clusters for resource management and container orchestration. Its orchestration system can obtain the appropriate resources satisfying the needs of requested services within defined Quality-of-Service (QoS) parameters, which is considered to be self-organised and self-managed meaning that users do not need to specifically request resource reservation. Nevertheless, this study has not shown insight into novel strategies of container orchestration for HPC systems.

Wrede et al. [94] performed their experiments on HPC clusters using Docker Swarm as the container orchestrator for automatic node scaling and using C++ algorithmic skeleton library Muesli [99] for load balance. Its proposed working environment is targeted for Cloud clusters. Usage of Docker cannot be easily extended to HPC infrastructures especially to HPC production systems due to the security risks.

4) Meta-Orchestration: Croupier [95] is a plugin implemented on Cloudify[9] server that is located at a separate node in addition to the nodes that are managed by an HPC workload manager and a Cloud orchestrator. Croupier establishes a *monitor* to collect the status of every infrastructure and the operations (e.g., status of the HPC batch queue). Croupier together with Cloudify, can orchestrate batch applications in both HPC and Cloud environments. Similarly, Di Nitto et al. [96] presented the SODALITE[10] framework by utilising XOpera[11] to manage the application deployment in heterogeneous infrastructures.

Colonnelli et al. [97] presented a proof-of-concept framework (i.e., Streamflow) to execute workflows on top of the hybrid architecture consisting of Kubernetes-managed Cloud and OCCAM [100] HPC cluster.

D. Section Highlights

HPC workload managers and Cloud orchestrators have distinct ways to manage clusters mainly because of their types of workloads and hardware resource availabilities. Table V summaries the differences of key features between HPC workload managers and Cloud Orchestrators. Typical HPC jobs are large workloads with long but ascertainable execution time and large throughput. HPC jobs are often submitted to a batch queue within a workload manager where jobs wait to be scheduled from minutes to days. *Per contra*, job requests can be granted immediately on Cloud as resources are available on demand. Batch-queuing is insufficient to satisfy the needs of Cloud communities: most of jobs are short in duration and the Cloud services are persistently long-running programs. Most of the HPC workload managers support *Checkpointing* that allows applications to save the execution states of a running job and restart the job from the checkpointing when a crash happens. This feature is critical for an HPC application with execution time typically from hours to months. Because it enables the application to recover from error states or resume from the state when it was previously terminated by the workload manager when its walltime limit had been reached or resource allocation had been exceeded. In contrary, jobs on Cloud, which are often micro-service programs, are usually relaunched in case of failures [101]. A container orchestrator offers an important property, i.e., container status monitoring. This is practical for long-running Cloud services, as it can monitor and replace unhealthy containers per desired configuration. HPC systems do not offer the equivalence of container *pod* which bundle performance monitoring services with the application itself as in Cloud systems [13]. Additionally, HPC workload managers often do not provide capabilities of application elasticity or necessary API at execution time, however, these capabilities are important for task migration and resource allocation changes at runtime on Cloud [102].

Section IV-C has reviewed the approaches to address the issues of container orchestration on HPC systems, which are summarised in Table VI. Overall, a container orchestrator on its own does not address all the requirements of HPC systems [3], as a result cannot replace existing workload managers in HPC centres. An HPC workload manager lacks micro-service support and deeply-integrated container management capabilities in which container orchestrators manifest their efficiency.

[9][Online]. Available: https://cloudify.co/

[10]SODALITE: SOftware-Defined AppLication Infrastructures managemenT and Engineering. https://www.sodalite.eu/

[11][Online]. Available: https://github.com/xlab-si/xopera-opera.

TABLE VII
OVERVIEW OF RESEARCH CHALLENGES AND POTENTIAL SOLUTIONS

Research challenges		Potential solutions	Open questions
Compatibility issues	Library compatibility	OS updates,low-level container runtime libraries	Reuse container images across platforms
	Compatibility of engines and images	Container standardisation (e.g. OCI)	
	Kernel optimisation	Using OS kernel to be library OS	
Security issues		Private container registry, namespace settings, OS updates, rootless installation of container engines, avoid root processes inside containers	Risk of using namespaces
Performance degradation		Trade-off between performance and portability	Leverage hardware resources without losing portability

V. RESEARCH CHALLENGES AND VISION

The distinctions between Cloud and HPC clusters are diminishing, especially with the trend of HPC Clouds in industry [103]. HPC Cloud is becoming an alternative to on-premise HPC clusters for executing scientific applications and business analytics models [16]. Containerisation technologies help to ease the efforts of moving applications between Cloud and HPC. Nevertheless, not all applications are suitable for containerisation. For instance, in the typical HPC applications such as weather forecast or modelling of computational fluid dynamics, any virtualisation or high-latency networks can become the bottlenecks for performance. Containerisation in HPC still faces challenges of different folds (Section V-A).

Interest in using containers on HPC systems is mainly due to the encapsulation and portability that yet may trade off with performance. In practice, containers deployed on HPC clusters often have large image size and as a result each HPC node can only host a few containers that are CPU-intensive and memory-demanding. In addition, implementation of AI frameworks such as TensorFlow and PyTorch [104] typically also have large container image size. Architecture of HPC containers should be able to easily integrate seamlessly with HPC workload managers. The research directions (Section V-B) which can be envisioned are not only to adapt the existing functionalities from Cloud to HPC, but to also explore the potentials of containerisation so as to improve the current HPC systems and applications.

A. Challenges and Open Issues

Although containerisation enables compatibility, portability and reproducibility, containerised environments still need to match the host architecture and exploit the underlying hardware. The challenges that containerisation faces on HPC systems are in three-fold: compatibility, security and performance. Some issues still remain as open questions. Table VII summarises the potential solutions to the research challenges and the open questions that will be discussed in this section.

1) Library Compatibility Issues: Mapping container libraries and their dependencies to the host libraries can cause incompatibility. Glibc [105], which is an implementation of C standard library that provides core supports and interfaces to kernel features, can be a common library dependency. The version of Glibc on the host may be older or newer than the one in the container image, consequently introducing symbol mismatches. Additionally, when the container OS (e.g., Ubuntu 18.04) and the host OS are different (e.g., CentOS 7), it is likely that some kernel ABI are incompatible, which may lead to container crashes or abnormal behaviours. This issue can also occur to MPI applications. As a result users must either build an exact version of the host MPI or have the privilege to mount the host MPI dependency path into the container.

A research direction to handle library mismatches between container images and hosts is to implement a container runtime library at a lower level. For instance, Nvidia implemented the library `libnvidia-container`[12] that manages driver or library matching at container runtime, i.e., using a hook interface to inject and/or activate the correct library versions. However, the `libnvidia-container` library can be only applied to Nividia GPUs. A significant modification of this library code is likely to be needed in order to be adapted for other GPU suppliers. In practice, such a compatibility layer would also require supports from different HPC interconnect and accelerator vendors.

2) Compatibility Issues of Container Engines and Images: Not all Docker images can be converted by HPC container engines to their own formats. Moreover, to reuse HPC container implementations between container engines, users need to learn different container command lines to build the corresponding images, which further complicates adoption of containers for HPC applications.

This issue calls for container standardisation. OCI is a Linux foundation project that designs open standards for container image formats (a `filesystem bundle` or `rootfs`) and *multiple data volume* [106]. Some guidelines were proposed in [63], i.e., a container should be:

- Not bound to higher-level frameworks, e.g., an orchestration stack;
- Not tightly associated with any particular vendor or project;
- Portable across a wide variety of OSs, hardware, CPUs, clusters, *etc*.

Unfortunately, this standard cannot guarantee that the runtime hooks built for one runtime can be used by another. For example,

[12] [Online]. Available: https://github.com/NVIDIA/libnvidia-container

container privileges (e.g., mount host filesystems) assumed by one container runtime may not be translated to unprivileged runtimes (e.g., not all HPC centres have `mount` namespace enabled) [107].

3) Kernel Optimisation: In general, containers are forbidden by the host to install their own kernel modules for the purpose of application isolation [108]. This is a limitation for the applications requiring kernel customisation, because the kernels of their HPC hosts cannot be tuned and optimised. Shen et al. [108] proposed an Xcontainer to address this issue by tuning the Linux kernel into library OS that supports binary compatibility. This functionality is yet to be explored in HPC containers.

4) Security Issues: Containers face three major threats [109]:

- Privilege Escalation. Attackers gain access to hosts and other containers by breaking out of their current containers.
- Denial-of-Service (DoS). An attack causes services to become inaccessible to users by disruption of a machine or network resources.
- Information Leak. Confidential details of other containers are leaked and utilised for further attacks.

Multiple or many containers share a host kernel, therefore, one container may infect other containers. In this case, a container does not reduce attack surfaces, but rather brings multiple instances of attack surfaces. For example, starting from version V3.0, Singularity has added `Cgroups` support that allows users to limit the resources consumed by containers without the help from a batch scheduling system (e.g., TORQUE). This feature helps to prevent DoS attacks when a container seizes control of all available system resources which prohibits other containers from operating properly.

Execution of HPC containers (including the Docker Engine starting from v19.03) does not require root privileges on the host. Containers in general adopt namespaces to isolate resources among users and map a root user inside a container to a non-root user on the host. The `User` namespace nevertheless is not a panacea to resolve all problems of resource isolation. `User` exposes code in the kernel to non-privileged users, which was previously limited to root users. A container environment is generated by users, and it is likely that some software inside a container may be embedded with security vulnerabilities. Root users inside a container may escalate their privileges via application level vulnerability. This can bring security issues to the kernel that does not account for mapped PIDs/GIDs. This issue can be addressed in two ways: (1) avoiding root processes inside HPC containers; (2) installing container engines with user permission instead of `sudo` installation. Security issues of the `user` namespace continue to be discovered even in the latest version of Linux kernels. Therefore, many HPC production centres have disabled the configuration of this namespace, which prevents usage of almost any state-of-the-art HPC containers. How to address the risks of using namespaces still remains an open question.

5) Performance Degradation: GPU and accelerators often require customised or proprietary libraries that need to be bound to container images so as to leverage performance. This operation is at the cost of portability [107]. It is *de facto* standard to utilise the optimised MPI libraries for HPC interconnects, such as InfiniBand and Slingshot [110], and it is likely that the container performance degrades in a different HPC infrastructure [22] (see Section III-B). There is no simple solution to address this issue.

Another example presented in [111] identified the performance loss due to increasing communication cost of MPI processes. This occurs when the number of containers (MPI processes running inside containers) rises on a single node, e.g., point to point communication (`MPI_Irecv, MPI_Isend`), polling of pending asynchronous messages (`MPI_Test`) and collective communication (`MPI_Allreduce`).

B. Research and Engineering Opportunities

Research studies should continue working on solutions to the open question identified in Section V-A. This section discusses current research and engineering directions that are interesting, yet still need further development. This section also identifies new research opportunities that yet need to be explored. The presentation of this section is arranged from short-term vision to long-term efforts. Table VIII summarises the potentials discovered in literature and the prospects given by the authors.

1) Containerisation of AI in HPC: Model training of AI/DL applications can immensely benefit from the compute power (GPU or CPU), storage and security [112] of HPC clusters in addition to the superior GPU-aware scheduling and features of workflow automation provided by workload managers. The trained models are subsequently deployed on Cloud for scalability at low cost and on HPC for computation speed. Exploiting HPC infrastructures for ML/DL training is becoming a topic of increasing importance [113]. For example, Fraunhofer[13] has developed the software framework Carme[14] that combines established open source ML and Data Science tools with HPC backends. The execution environments of the tools are provided by predefined Singularity containers.

AI applications are usually developed with high-level scripting languages or frameworks, e.g., TensorFlow and PyTorch, which often require connections to external systems to download a list of open-source software packages during execution. For instance, an AI application written in Python cannot be compiled into an executable that has included all the dependencies ready for execution as in C/C++. Therefore, the developers need flexibility to customise the execution environments. Since HPC environments, especially on HPC production systems, are often based on closed-source applications and their users have restricted account privileges and security restrictions [6], deployment of AI applications on HPC infrastructures is challenging. Besides the predefined module environments or virtual environments (such as Anaconda), containerisation can be an alternative candidate, which enables easy transition of AI workloads to HPC while fully taking advantage of HPC hardware and the optimised libraries of AI applications without compromising security. Huerta et al. [114] recommend three guidelines for containerisation of AI applications for HPC centres:

[13]Fraunhofer: A German research organisation. https://www.fraunhofer.de/

[14]https://www.itwm.fraunhofer.de/en/departments/hpc/data-analysis-and-machine-learning/carme-softwarestack.html

TABLE VIII
FUTURE DIRECTIONS OF RESEARCH AND ENGINEERING

Topics	Importance	State-of-art trends	Prospects given by the authors
Containerisation of AI in HPC	Leverage HPC systems for ML/DL training	Containerised AI apps & frameworks	Improve scalability; Enable out-of-box usage
HPC container registry	Pre-build images accessible within HPC centres, ensure container security	HPC centres set up private registries	WLMs boot containers from registries without users awareness
Linux namespace guideline	Ensure security	HPC centres provide namespace guidelines	Different user groups to have different set of namespaces enabled
DevOps	Research reproducibility	Integration of Singularity with Jenkins	HPC-specific DevOps tools
Middleware system	Flexible and easy to plugin or plugout new components	Transfer Docker to HPC containers and perform the deployment onto HPC systems	Enable DevOps on HPC systems
Resource elasticity	Flexible usage of hardware resources	Kubernetes to instantiate the containerised HPC schedulers	Integration of containers to introduce resource elasticity to WLM
Moving toward minimal OS	Reduce maintenance efforts	–	Maintain minimal OS kernel and containerised the rest of the HPC software stack

- Provide up-to-date documentation and tutorials to set up or launch containers.
- Maintain versatile and up-to-date base container images that users can clone and adapt, such as a container registry (see Section V-B2).
- Give instructions on installation or updates of software packages into containers. The AI program depends on distributed training software, such as Horovod [115], which then depends on system architecture and specific versions of software packages such as MPI.

Increasing amount of new software frameworks are being developed using containerisation technologies to facilitate deployment of AI applications on HPC systems. Further research is still needed to improve scalability and enable out-of-box usage.

2) HPC Container Registry: Container registry is a useful repository to provide pre-built container images that can be accessed easily either by public or private users by pulling images to the host directly. It is portable to deploy applications in this way on Cloud clusters. Accesses to external networks are often blocked in HPC centres, so users need to upload images onto the clusters manually. One solution is to set up a private registry within the HPC centres that offer pre-built images suitable for the targeted systems and architectures.

A container registry is also a way to ensure container security. It is a good security practice to ensure that images executed on the HPC systems are signed and pulled from a trusted registry. Scanning vulnerabilities on the registry should be regularly performed.

To simplify usage, the future work can enable HPC workload managers to boot the default containers on the compute nodes (by pulling images from the private registry) which match the environments with all the required libraries and configuration files of user login nodes where users implement their own workflows and submit their jobs. The jobs should be started without user awareness of the presence of containers and without additional user intervention.

3) Linux Namespace Guidelines: The set of Linux namespaces used within an implementation depends on the policies of HPC centres [116]. HPC centres should provide clear instructions on the availabilities of namespaces. For example, different user groups may have different namespaces enabled or disabled. A minimal set of namespaces should be enabled for a general user group: `mount` and `user`, which are suitable for node-exclusive scheduling. `PID` and `Cgroups` should be provided to restrict resource usage and enforce process privacy, which are useful for shared-node scheduling. Advanced use cases may require additional sets of namespaces. When users submit the container jobs, workload managers can start the containers with appropriate namespaces enabled.

4) DevOps: DevOps aims at integrating efforts of development (Dev) and operations (Ops) to automate fast software delivery while ensuring correctness and reliability [117], [118]. This concept is influential in Cloud Computing and has been widely adopted in industry, as DevOps tools minimise the overhead of managing a large amount of micro-services. In HPC environments, typical applications have large workloads, hence

the usage of DevOps should concentrate on research reproducibility. Nevertheless, the off-the-shelf DevOps tools are not well fitted for HPC environments, e.g., the dependencies of MPI applications are too foreign for the state-of-the-art DevOps tools. A potential solution is to develop HPC-specific DevOps tools for the applications that are built and executed on on-premise clusters [16]. Unfortunately, HPC environments are known to be inflexible and typical HPC applications are optimised to leverage resources, thereby generation of DevOps workflows can be restricted and slow. Such obstacles can be overcome by containerisation, which may provision DevOps environments. For instance, Sampedro et al. [119] integrate Singularity with Jenkins [120] that brings CICD[15] practices into HPC workflows. Jenkins is an open-source automation platform for building and deploying software, which has been applied at some HPC sites as a general-purpose automation tool.

5) Middleware System: A middleware system, which bridges container building environments with HPC resource managers and schedulers, can be flexible. A middleware system can be either located on an HPC cluster or connect to it with secured authentication. The main task of the middleware is to perform job deployment, job management, data staging and generating non-root container environments [121]. Different container engines can be swiftly switched, optimisation mechanisms can be adapted to the targeted HPC systems and workflow engines [122] can be easily plugged in. Middleware systems can be a future research direction that provides a portable way to enable DevOps in HPC centres.

6) Resource Elasticity: One major difference between resource management on HPC and Cloud is the elasticity [123], i.e., an HPC workload manager runs on a fixed set of hardware resources and the workloads of its jobs at any point can not exceed the resource capacity, while Cloud orchestrators can scale up automatically the hardware resources to satisfy user needs (e.g., AWS spot instances). Static reservation is a limitation for efficient resource usages on HPC systems [124]. One future direction of containerisation for HPC systems can work towards improvement of the elasticity of HPC infrastructure, which can be introduced to its workload manager. In [123], the authors presented a novel architecture that utilises Kubernetes to instantiate the containerised HPC workload manager. In this way, the HPC infrastructure is dynamically instantiated on demand and can be served as a single-tenant or multi-tenant environment. A complete containerised environments on HPC system may be impractical and much more exploration is still needed.

7) Moving Towards Minimal OS: Containers may be utilised to partially substitute the current HPC software stack. Typical compute nodes on HPC clusters do not contain local storage (e.g., hardware disk), therefore lose states after reboots. The compute node boots via a staged approach [116]: (1) a kernel and initial RAM disk are loaded via a network device; (2) a root filesystem is mounted via the network. In a monolithic stateless system, modification of the software components often requires system rebooting to completely activate the functions of updates. Using containerised software packages on top of a minimal OS (base image) on the compute nodes, reduces the number of components in the kernel image, hence decreasing the frequency of node reboots. Furthermore, the base image of reduced size also simplifies the post-boot configurations that need to run in the OS image itself, consequently the node rebooting time is minimised. Additionally, when a failure occurs, a containerised service can be quickly replaced without affecting the entire system. Long-term research is required on HPC workload managers to control the software stack and workloads that are partially native and partially containerised. Moreover, it needs to explored whether containerisation of the entire OS on HPC systems is feasible.

[15]CICD: Continuous integration, delivery and deployment. It is widely used in DevOps communities.

VI. Concluding Remarks

This paper presents a survey and taxonomy for the state-of-the-art container engines and container orchestration strategies specifically for HPC systems. It underlines differences of containerisation on Cloud and HPC systems. The research and engineering challenges are also discussed and the opportunities are envisioned.

HPC systems start to utilise containers as thereof reduce environment complexity. Efforts have been also made to ameliorate container security on HPC systems. This article identified three points to increase the security level: (1) set on-site container registry, (2) give Linux namespaces guidelines (3) and remove root privilege meanwhile avoid permission escalation. Ideally, HPC containers should require no pre-installation of container engines or installation can be performed without root privileges, which not only meets the HPC security requirements but also simplifies the container usability.

Containers will continue to play a role in reducing the performance gap and deployment complexity between on-premise HPC clusters and public Clouds. Together with the advancement of low-latency networks and accelerators (e.g., GPUs, TPUs [125]), it may eventually reshape the two fields. Containerised workloads can be moved from HPC to Cloud so as to temporarily relieve the peak demands and can be also scheduled from Cloud to HPC in order to exploit the powerful hardware resources. The research and engineering trend are working towards implementation of the present container orchestrators within HPC clusters, which however still remains experimental. Many studies have been devoted to container orchestration on Cloud, however, it can be foreseen that the strategies will be eventually introduced to HPC workload managers.

In the future, it can be presumed that containerisation will play an essential role in application development, improve resource elasticity and reduce complexity of HPC software stacks.

Acknowledgments

The authors would like to thank Dr. Joseph Schuchart for proof-reading the contents.

References

[1] M. Khan, T. Becker, P. Kuppuudaiyar, and A. C. Elster, "Container-based virtualization for heterogeneous HPC clouds: Insights from the EU H2020 CloudLightning project," in *Proc. IEEE Int. Conf. Cloud Eng.*, Piscataway, NJ, USA, 2018, pp. 392–397.

[2] M. A. Rodriguez and R. Buyya, "Container-based cluster orchestration systems: A taxonomy and future directions," *Softw.: Pract. Experience*, vol. 49, no. 5, pp. 698–719, 2019.

[3] L. Abdollahi Vayghan, M. A. Saied, M. Toeroe, and F. Khendek, "Deploying microservice based applications with kubernetes: Experiments and lessons learned," in *Proc. IEEE 11th Int. Conf. Cloud Comput.*, (Piscataway, New Jersey, US), 2018, pp. 970–973.

[4] J. M. Olivier Terzo, ed., *HPC, Big Data, and AI Convergence Towards Exascale: Challenge and Vision*, 1st ed.. Boca Raton, FL, USA: CRC Press, Jan. 2022.

[5] R. McLay, K. W. Schulz, W. L. Barth, and T. Minyard, "Best practices for the deployment and management of production HPC clusters," in *Proc. IEEE Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2011, pp. 1–11.

[6] D. Brayford, S. Vallecorsa, A. Atanasov, F. Baruffa, and W. Riviera, "Deploying AI frameworks on secure HPC systems with containers," in *Proc. IEEE High Perform. Extreme Comput. Conf.*, 2019, pp. 1–6.

[7] G. Yi and V. Loia, "High-performance computing systems and applications for AI," *J. Supercomputing*, vol. 75, pp. 4248–4251, 06 2019.

[8] E. Casalicchio, "Autonomic orchestration of containers: Problem definition and research challenges," in *Proc. 10th EAI Int. Conf. Perform. Eval. Methodol. Tools 10th EAI Int. Conf. Perform. Eval. Methodol. Tools*, 2017, Art. no. 287C290,.

[9] A. Tosatto, P. Ruiu, and A. Attanasio, "Container-based orchestration in cloud: State of the art and challenges," in *Proc. IEEE 9th Int. Conf. Complex Intell. Softw. Intensive Syst.*,, 2015, Art. no. 70C75.

[10] C. Pahl, A. Brogi, J. Soldani, and P. Jamshidi, "Cloud container technologies: A state-of-the-art review," *IEEE Trans. Cloud Comput.*, vol. 7, no. 3, pp. 677–692, Third Quarter 2017.

[11] E. Casalicchio, "Container Orchestration: A Survey," in *Systems Modeling: Methodologies and Tools*, A. Puliafito and K. S. Trivedi, eds., 2019, pp. 221–235.

[12] N. Nguyen and D. Bein, "Distributed MPI cluster with docker swarm mode," in *Proc. IEEE 7th Annu. Comput. Commun. Workshop Conf.*, 2017, pp. 1–7.

[13] D. Bernstein, "Containers and cloud: From LXC to docker to kubernetes," *IEEE Cloud Comput.*, vol. 1, no. 3, pp. 81–84, Sep. 2014.

[14] Z. Zhong and R. Buyya, "A cost-efficient container orchestration strategy in kubernetes-based cloud computing infrastructures with heterogeneous resources," *ACM Trans. Internet Technol.*, vol. 20, pp. 1–14, Apr. 2020.

[15] E. Casalicchio and S. Iannucci, "The state-of-the-art in container technologies: Application, orchestration and security," *Concurrency Comput.: Pract. Experience*, vol. 32, 2019, Art. no. e5668.

[16] M. A. S. Netto, R. N. Calheiros, E. R. Rodrigues, R. L. F. Cunha, and R. Buyya, "HPC cloud for scientific and business applications: Taxonomy, vision, and research challenges," *ACM Comput. Surveys*, vol. 51, pp. 1–29, Jan. 2018.

[17] D. Merkel, "Docker: Lightweight linux containers for consistent development and deployment," *Linux J.*, vol. 2014, pp. 76–90, Mar. 2014.

[18] J. S. Hale, L. Li, C. N. Richardson, and G. N. Wells, "Containers for portable, productive, and performant scientific computing," *Comput. Sci. Eng.*, vol. 19, pp. 40–50, Nov. 2017.

[19] P. Sharma, L. Chaufournier, P. Shenoy, and Y. C. Tay, "Containers and virtual machines at scale: A comparative study," in *Proc. 17th Int. Middleware Conf.*,, 2016, pp. 1–13.

[20] R. Morabito, J. Kj0Ellman, and M. Komu Hypervisors, "Lightweight virtualization: A performance comparison," in *Proc. IEEE Int. Conf. Cloud Eng.*, 2015, pp. 386–393.

[21] "Containers on Virtual Machines or Bare Metals?," *Tech. Rep.*, VMWare, VMware, Inc. 3401 Hillview Avenue Palo Alto, CA, USA, Dec. 2018.

[22] O. Rudyy, M. Garcia-Gasulla, F. Mantovani, A. Santiago, R. Sirvent, and M. Vzquez, "Containers in HPC: A scalability and portability study in production biological simulations," in *Proc. IEEE Int. Parallel Distrib. Process. Symp.*, 2019, pp. 567–577.

[23] G. Hu, Y. Zhang, and W. Chen, "Exploring the performance of singularity for high performance computing scenarios," in *Proc. IEEE 21st Int. Conf. High Perform. Comput. Commun. IEEE 17th Int. Conf. Smart City, IEEE 5th Int. Conf. Data Sci. Syst.*, 2019, pp. 2587–2593.

[24] 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 *Proc. IEEE Int. Conf. Cloud Comput. Technol. Sci.*, 2017, pp. 74–81.

[25] J. Zhang, X. Lu, and D. K. Panda, "Is singularity-based container technology ready for running MPI applications on HPC clouds?," in *Proc. 10th Int. Conf. Utility Cloud Comput.*, 2017, pp. 151–160.

[26] J. P. Martin, A. Kandasamy, and K. Chandrasekaran, "Exploring the support for high performance applications in the container runtime environment," *Hum.-Centric Comput. Inf. Sci.*, vol. 8, pp. 1–15, Dec. 2018.

[27] S. Abraham, A. K. Paul, R. I. S. Khan, and A. R. Butt, "On the use of containers in high performance computing environments," in *Proc. IEEE 13th Int. Conf. Cloud Comput.*, 2020, pp. 284–293.

[28] R. Priedhorsky and T. Randles, "Charliecloud: Unprivileged containers for user-defined software stacks in HPC," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2017, pp. 1–10.

[29] S.K. S., *Practical LXC and LXD: Linux Containers for Virtualization and Orchestration*, 1st ed. USA: Apress, 2017.

[30] P. Saha, A. Beltre, P. Uminski, and M. Govindaraju, "Evaluation of docker containers for scientific workloads in the cloud," in *Proc. Pract. Experience Adv. Res. Comput.*, 2018, pp. 1–8.

[31] L. Gerhardt et al., "Shifter: Containers for HPC," in *Proc. J. Phys.: Conf. Ser.*, 2017, Art. no. 082021.

[32] W. Gropp, E. Lusk, and A. Skjellum, *Using MPI: Portable Parallel Programming With the Message-Passing Interface*. Cambridge, MA, USA: MIT Press, 1994.

[33] Message Passing Interface Forum, "MPI: A message-passing interface standard," Jun. 2021.

[34] G. M. Kurtzer, V. V. Sochat, and M. Bauer, "Singularity: Scientific containers for mobility of compute," *PLoS One*, vol. 12, 2017, Art. no. e0177459.

[35] L. Benedicic, F. A. Cruz, A. Madonna, and K. Mariotti, "Sarus: Highly scalable docker containers for HPC systems," in *High Performance Computing*, M. Weiland, G. Juckeland, S. Alam, and H. Jagode, eds., Berlin, Germany: Springer, 2019, pp. 46–60.

[36] S. GantikowWalter and C. Reich, "Rootless containers with podman for HPC," in *High Performance Computing*, H. H. Jagode, G. AnztJuckeland, and H. Ltaief, eds., Berlin, Germany:, Springer, 2020, pp. 343–354.

[37] K. Hightower, B. Burns, and J. Beda, *Kubernetes: Up and Running Dive Into the Future of Infrastructure*, 1st ed.. Sebastopol, California, US: O'Reilly Media, Inc., 2017.

[38] A. Ruhela et al., "Containerization on petascale HPC clusters," in *Proc. HPC, State Pract. talk Int. Conf. High Perform.*, Texas Scholar Works, Nov. 2020.

[39] A. Azab, "Enabling docker containers for high-performance and many-task computing," in *Proc. IEEE Int. Conf. Cloud Eng.*, 2017, pp. 279–285.

[40] D. Bahls, "Evaluating shifter for HPC applications," in *Proc. Cray User Group Conf.*, 2016.

[41] G. MuscianisiFiameni and A. Azab, "Singularity GPU Containers Execution on HPC Cluster," in *High Performance Computing*, M.G. WeilandJuckeland, S. Alam, and H. Jagode, eds., Berlin, Germany: Springer, 2019, pp. 61–68.

[42] IMB, "Introducing intel MPI benchmarks," 2022. [Online]. Available: https://www.intel.com/content/www/us/en/developer/articles/technical/intel-mpi-benchmarks.html

[43] J. Dongarra, M. A. Heroux, and P. Luszczek, "HPCG benchmark: A new metric for ranking high performance computing systems," Knoxville, Tennessee, vol. 42, 2015.

[44] Cray XC 40, "Overviw of cray XC40 architecture," 2022. [Online]. Available: https://www.alcf.anl.gov/files/CrayXC40Brochure.pdf

[45] J. J. Dongarra, C. B. Moler, J. R. Bunch, and G. W. Stewart, *LINPACK Users' Guide*. Philadelphia: Society for Industrial and Applied Mathematics, 1979.

[46] NAMD "Simulation for molecular dynamics," 2022. [Online]. Available: https://www.ks.uiuc.edu/Research/namd/

[47] VASP, "Atomic scale materials modelling," 2022. [Online]. Available: https://www.hpc.cineca.it/content/vasp-benchmark

[48] WRF, "Weather research and forecasting model," 2022, [Online]. Available: https://openbenchmarking.org/test/pts/wrf-1.0.0

[49] AMBER, "Assisted model building with energy refinement," 2022. [Online]. Available: http://ambermd.org/doc12/Amber18.pdf

[50] M. Abadi et al., "TensorFlow: A system for large-scale machine learning," in *Proc. 12th USENIX Conf. Operating Syst. Des. Implementation*, 2016, pp. 265–283.

[51] HPGMG, "High-performance geometric multigrid, Github," 2022. [Online]. Available: https://github.com/hpgmg/hpgmg
[52] M. Hovestadt, O. Kao, A. Keller, and A. Streit, "Scheduling in HPC Resource Management Systems: Queuing versus Planning," in *Job Scheduling Strategies for Parallel Processing*, D. Feitelson, L. Rudolph, and U. Schwiegelshohn, eds., Berlin, Germany: Springer, 2003, pp. 1–20.
[53] D. Klusáček, V. Chlumský, and H. Rudová, "Planning and optimization in torque resource manager," in *Proc. 24th Int. Symp. High- Perform. Parallel Distrib. Comput.*, 2015, pp. 203–206.
[54] G. Staples, "Torque resource manager," in *Proc. IEEE/ACM Conf. Supercomputing*, 2006, pp. 8–es.
[55] S. Zhou, X. Zheng, J. Wang, and P. Delisle, "Utopia: A load sharing facility for large, heterogeneous distributed computer systems," *Softw. Pract. Experience*, vol. 23, 1993, Art. no. 1305C1336.
[56] W. Gentzsch, "Sun grid engine: Towards creating a compute power grid," in *Proc. IEEE/ACM 1st Int. Symp. Cluster Comput. Grid*, 2001, pp. 35–36.
[57] N. Capit et al., "A batch scheduler with high level components," in *Proc. IEEE 5th Int. Symp. Cluster Comput. Grid*, 2005, pp. 776–783.
[58] M. A. Jette, A. B. Yoo, and M. Grondona, "SLURM: Simple Linux Utility for Resource Management," in *Proceedings of Job Scheduling Strategies for Parallel Processing*, Berlin, Germany: Springer, 2003, pp. 44–60.
[59] D. Jackson, Q. Snell, and M. Clement, "Core Algorithms of the Maui Scheduler," in *Job Scheduling Strategies for Parallel Processing* (D. G. Feitelson and L. Rudolph, eds.), Berlin, Germany: Springer, 2001, pp. 87–102.
[60] S. Prabhakaran, Dynamic resource management and job scheduling for high performance computing, PhD thesis, Technische Universität Darmstadt, Darmstadt, Aug. 2016.
[61] A. W. Mu'alem and D. G. Feitelson, "Utilization, predictability, workloads, and user runtime estimates in scheduling the IBM SP2 with backfilling," *IEEE Trans. Parallel Distrib. Syst.*, vol. 12, no. 6, pp. 529–543, Jun. 2001.
[62] D. G. Feitelson, L. Rudolph, and U. Schwiegelshohn, "Parallel Job Scheduling — A Status Report," in Job Scheduling Strategies for Parallel Processing (D. G. Feitelson, L. Rudolph, and U. Schwiegelshohn, eds.), Berlin, Germany: Springer Berlin, 2005, pp. 1–16.
[63] A. Khan, "Key characteristics of a container orchestration platform to enable a modern application," *IEEE Cloud Comput.*, vol. 4, no. 9, pp. 42–48, Sep. 2017.
[64] S. Pandey and V. Tokekar, "Prominence of MapReduce in Big Data processing," in *Proc. IEEE 4th Int. Conf. Commun. Syst. Netw. Technol.*, 2014, pp. 555–560.
[65] M. Zaharia et al., "Apache spark: A unified engine for Big Data processing," *Commun. ACM*, vol. 59, pp. 56–65, Oct. 2016.
[66] N. Narkhede, G. Shapira, and T. Palino, *Kafka: The Definitive Guide Real-Time Data and Stream Processing At Scale*, 1st ed.Sebastopol, California, US: O'Reilly Media, Inc., 2017.
[67] G. Kandiraju, H. Franke, M. D. Williams, M. Steinder, and S. M. Black, "Software defined infrastructures," *IBM J. Res. Dev.*, vol. 58, pp. 2:1–2:13, Mar. 2014.
[68] P. X. Gao et al., "Network requirements for resource disaggregation," in *Proc. 12th USENIX Conf. Operating Syst. Des. Implementation Assoc.*, 2016, Art. no. 249C264.
[69] F. Soppelsa and C. Kaewkasi, *Native Docker Clustering With Swarm*. Birmingham, UK: Packt Publishing Ltd, 2016.
[70] B. Hindman et al., "Mesos: A platform for fine-grained resource sharing in the data center," in *Proc. 8th USENIX Conf. Networked Syst. Des. Implementation Assoc.*, 2011, Art. no. 295C308.
[71] T. White, *Hadoop: The Definitive Guide*. Sebastopol, California: O'Reilly Media, Inc., 2012.
[72] V. K. Vavilapalli et al., "Apache hadoop YARN: Yet another resource negotiator," in *Proc. 4th Annu. Symp. Cloud Comput.*, 2013, pp. 1–16.
[73] G. Sammons, *Exploring Ansible 2: Fast and Easy Guide*. North Charleston, SC, USA: CreateSpace Independent Publishing Platform, 2016.
[74] O. Sefraoui, M. Aissaoui, and M. Eleuldj, "OpenStack: Toward an open-source solution for cloud computing," *Int. J. Comput. Appl.*, vol. 55, pp. 38–42, 2012.
[75] S. S. Manvi and G. K. Shyam, "Resource management for infrastructure as a service (IaaS) in cloud computing: A survey," *J. Netw. Comput. Appl.*, vol. 41, pp. 424–440, 2014.
[76] W. Sun, K. Zhang, S.-K. Chen, X. Zhang, and H. Liang, "Software as a Service: An Integration Perspective," in *Service-Oriented Computing*, B.J. Krämer, K.-J. Lin, and P. Narasimhan, eds., Berlin, Germany: Springer, 2007, pp. 558–569.
[77] T. Rosado and J. Bernardino, "An overview of openstack architecture," in *Proc. 18th Int. Database Eng. Appl. Symp.*, 2014, Art. no. 366C367.
[78] G. P. Fernandez and A. Brito, "Secure container orchestration in the cloud: Policies and implementation," in *Proc. 34th ACM/SIGAPP Symp. Appl. Comput.*, 2019, Art. no. 138C145.
[79] P.-J. Maenhaut, B. Volckaert, V. Ongenae, and F. De Turck, "Resource management in a containerized cloud: Status and challenges," *J. Netw. Syst. Manage.*, vol. 28, pp. 197–246, 11 2019.
[80] R. Buyya and S.N. Srirama, *A Lightweight Container Middleware for Edge Cloud Architectures*, Hoboken, NJ, USA: Wiley, 2019, pp. 145–170.
[81] T. S. Somasundaram and K. Govindarajan, "CLOUDRB: A framework for scheduling and managing high-performance computing (HPC) applications in science cloud," *Future Gener. Comput. Syst.*, vol. 34, pp. 47–65, 2014.
[82] K. Cho, H. Lee, K. Bang, and S. Kim, "Possibility of HPC application on cloud infrastructure by container cluster," in *Proc. IEEE Int. Conf. Comput. Sci. Eng. IEEE Int. Conf. Embedded Ubiquitous Comput.*, 2019, pp. 266–271.
[83] C. Evangelinos and C. Hill, "Cloud computing for parallel scientific HPC applications: Feasibility of running coupled atmosphere-ocean climate modelson amazons EC2," *CCA*, pp. 22–34, Oct. 2008.
[84] A. M. Beltre, P. Saha, M. Govindaraju, A. Younge, and R. E. Grant, "Enabling HPC workloads on cloud infrastructure using kubernetes container orchestration mechanisms," in *Proc. IEEE/ACM Int. Workshop Containers New Orchestration Paradigms Isolated Environments*, 2019, pp. 11–20.
[85] Q. Wofford, P. G. Bridges, and P. Widener, "A layered approach for modular container construction and orchestration in HPC environments," in *Proc. 11th Workshop Sci. Cloud Comput.*, 2021, Art. no. 1C8.
[86] S. Julian, M. Shuey, and S. Cook, "Containers in research: Initial experiences with lightweight infrastructure," in *Proc. XSEDE16 Conf. Diversity Big Data Sci.*, 2016, pp. 1–6.
[87] J. Higgins, V. Holmes, and C. Venters, "Orchestrating docker containers in the HPC environment," in *High Performance Computing*, J. M. Kunkel and T. Ludwig, eds., Berlin, Germany: Springer, 2015, pp. 506–513.
[88] N. Zhou, Y. Georgiou, L. Zhong, H. Zhou, and M. Pospieszny, "Container orchestration on HPC systems," in *Proc. IEEE Int. Conf. Cloud Comput.*, 2020, pp. 34–36.
[89] N. Zhou et al., "Container orchestration on HPC systems through kubernetes," *J. Cloud Comput.: Adv. Syst. Appl.*, vol. 10, pp. 1–4, 2021.
[90] N. Zhou, "Containerization and orchestration on HPC systems," in *Sustained Simulation Performance*, Berlin, Germany: Springer, 2021.
[91] N. Zhou et al., *CYBELE: A. Hybrid Architecture for HPC and Big Data for AI Applications in Agriculture*, ch. 13. Boca Raton, FL, USA: CRC Press, 2022.
[92] F. Liu, K. Keahey, P. Riteau, and J. Weissman, "Dynamically negotiating capacity between on-demand and batch clusters," in *Proc. IEEE Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2018, pp. 493–503.
[93] E. Piras, L. Pireddu, M. Moro, and G. Zanetti, "Container Orchestration on HPC Clusters," in *High Performance Computing*, M.G. WeilandJuckeland, S. Alam, and H. Jagode, eds., Berlin, Germany: Springer, 2019, pp. 25–35.
[94] F. Wrede and V. von Hof, "Enabling efficient use of algorithmic skeletons in cloud environments: Container-based virtualization for hybrid CPU-GPU execution of data-parallel skeletons," in *Proc. Symp. Appl. Comput.*, 2017, Art. no. 1593C1596.
[95] J. Carnero and F. J. Nieto, "Running simulations in HPC and cloud resources by implementing enhanced TOSCA workflows," in *Proc. IEEE Int. Conf. High Perform. Comput. Simul.*, 2018, pp. 431–438.
[96] E. Di Nitto et al., "An approach to support automated deployment of applications on heterogeneous Cloud-HPC infrastructures," in *Proc. IEEE 22nd Int. Symp. Symbolic Numeric Algorithms Sci. Comput.*, 2020, pp. 133–140.
[97] I. Colonnelli, B. Cantalupo, I. Merelli, and M. Aldinucci, "StreamFlow: Cross-breeding cloud with HPC," *IEEE Trans. Emerg. Topics Comput.*, vol. 9, no. 4, pp. 1723–1737, Fourth Quarter 2021.
[98] "MoabHPC Suite," Accessed: Jul. 08, 2020. [Online]. Available: https://support.adaptivecomputing.com/wp-content/uploads/2019/06/Moab-HPC-Suite_datasheet_20190611.pdf
[99] P. Ciechanowicz, M. Poldner, and H. Kuchen, "The münster skeleton library muesli: A comprehensive overview," Tech. Rep., University of Mnster, European Research Center for Information Systems Münster, Germany, 2009.

[100] M. Aldinucci, S. Bagnasco, S. Lusso, P. Pasteris, S. Rabellino, and S. Vallero, "OCCAM: A flexible, multi-purpose and extendable HPC cluster," *J. Phys. Conf. Ser.*, vol. 898, 2017, Art. no. 082039.
[101] A. Reuther et al., "Scalable system scheduling for HPC and Big Data," *J. Parallel Distrib. Comput.*, vol. 111, pp. 76–92, 2018.
[102] A. Souza, M. Rezaei, E. Laure, and J. Tordsson, "Hybrid resource management for HPC and data intensive workloads," in *Proc. IEEE/ACM 19th Int. Symp. Cluster*, 2019, pp. 399–409.
[103] P. Marshall, H. Tufo, and K. Keahey, "High-performance computing and the cloud: A. match made in heaven or hell?," *XRDS: Crossroads ACM Mag. Students*, vol. 19, 2013, Art. no. 52C57.
[104] A. Paszke et al., "PyTorch: An imperative style, high-performance deep learning library," in *Proc. Adv. Neural Inf. Process. Syst. 32: Annu. Conf. Neural Inf. Process. Syst.*, 2019, pp. 8024–8035.
[105] B. Gough, *GNU Scientific Library Reference Manual - Third Edition. Network Theory Ltd*, 3rd ed.. Boca Raton, FL, USA: CRC Press, 2009.
[106] S. Nadgowda, S. Suneja, N. Bila, and C. Isci, "Voyager: Complete container state migration," in *Proc. IEEE 37th Int. Conf. Distrib. Comput. Syst.*, 2017, pp. 2137–2142.
[107] R. S. Canon and A. Younge, "A case for portability and reproducibility of HPC containers," in *Proc. IEEE/ACM Int. Workshop Containers New Orchestration Paradigms Isolated Environments HPC*, 2019, pp. 49–54.
[108] Z. Shen et al., "X-containers: Breaking down barriers to improve performance and isolation of cloud-native containers," in *Proc. 24th Int. Conf. Architectural Support Program. Lang. Operating Syst.*, 2019, Art. no. 121C135.
[109] H. Gantikow, C. Reich, M. Knahl, and N. Clarke, "Providing security in container-based HPC runtime environments," in *High Performance Computing*, M. Taufer, B. Mohr, and J. M. Kunkel, eds., Berlin, Germany: Springer: 2016, pp. 685–695.
[110] K. Shafie Khorassani, C. C. Chen, B. Ramesh, A. Shafi, H. Subramoni, and D. Panda, "High performance MPI over the slingshot interconnect: Early experiences," in *Practice and Experience in Advanced Research Computing*, New York, NY, USA: Association for Computing Machinery, 2022.
[111] J. Zhang, X. Lu, and D. K. Panda, "High performance MPI library for container-based HPC cloud on InfiniBand clusters," in *Proc. IEEE 45th Int. Conf. Parallel Process.*, 2016, pp. 268–277.
[112] G. Mateescu, W. Gentzsch, and C. J. Ribbens, "Hybrid computing-where HPC meets grid and cloud computing," *Future Gener. Comput. Syst.*, vol. 27, 2011, Art. no. 440C453.
[113] R. Mayer and H.-A. Jacobsen, "Scalable deep learning on distributed infrastructures: Challenges, techniques, and tools," *ACM Comput. Surveys*, vol. 53, pp. 1–3, 2020.
[114] E. A. Huerta et al., "Convergence of artificial intelligence and high performance computing on NSF-supported cyberinfrastructure," *J. Big Data*, vol. 7, pp. 1–12, 2020.
[115] A. Sergeev and M. D. Balso, "Horovod: Fast and easy distributed deep learning in TensorFlow," 2018, *arXiv:1802.05799*.
[116] B. S. Allen et al., "Modernizing the HPC system software stack," 2020, *arXiv:2007.10290*.
[117] M. Hüttermann, *DevOps for Developers*. New York, NY, USA: Apress, 2012.
[118] L. Leite, C. Rocha, F. Kon, D. Milojicic, and P. Meirelles, "A survey of DevOps concepts and challenges," *ACM Comput. Surveys*, vol. 52, pp. 1–35, 2019.
[119] Z. Sampedro, A. Holt, and T. Hauser, "Continuous Integration and Delivery for HPC: Using Singularity and Jenkins," in *Proc. Practi. Experience on Adv. Res. Comput.*, New York, NY, USA: Association for Computing Machinery, 2018.
[120] J. Muli and A. Okoth, *Jenkins Fundamentals: Accelerate Deliverables, Manage Builds, and Automate Pipelines With Jenkins*. Birmingham, UK: Packt Publishing, 2018.
[121] M. H02b and D. Kranzlmller, "Enabling EASEY deployment of containerized applications for future HPC systems," in *Lecture Notes in Computer Science*, Berlin, Germany: Springer, 2020, pp. 206–219.
[122] M. Barika, S. Garg, A. Y. Zomaya, L. Wang, A. V. Moorsel, and R. Ranjan, "Orchestrating Big Data analysis workflows in the cloud: Research challenges, survey, and future directions," *ACM Comput. Surv.*, vol. 52, pp. 1–41, 2019.
[123] C. Crin, N. Greneche, and T. Menouer, "Towards pervasive containerization of HPC job schedulers," in *Proc. IEEE 32nd Int. Symp. Comput. Architecture High Perform. Comput.*, 2020, pp. 281–288.
[124] D. Huber, M. Streubel, I. Comprés, M. Schulz, M. Schreiber, and H. Pritchard, "Towards dynamic resource management with MPI sessions and PMIx," in *Proc. 29th Eur.*, 2022, Art. no. 57C67.
[125] N. P. Jouppi et al., "In-datacenter performance analysis of a tensor processing unit," in *Proc. 44th Annu. Int. Symp. Comput. Architecture*, 2017, Art. no. 1C12.

Naweiluo Zhou received the PhD degree in computer science from Grenoble Alpes University (France) and INRIA (French Institute for Research in Computer Science and Automation). Zhou is a research scientist in Leibniz Supercomputing Centre (LRZ, Munich Germany), who also worked in High Performance Computing Center Stuttgart (HLRS, Stuttgart Germany). Zhou's research interest is exploiting HPC and Cloud technologies to address the challenges in different domain science.

Huan Zhou received the PhD degree in engineering from the University of Stuttgart, Germany. She is a research scientist working in HLRS. Her research interests include parallel and distributed computation in software engineering, performance analysis/modelling and parallel programming models. She has published in various journals and conferences in her field including parallel computing, ICPP, cloud computing.

Dennis Hoppe received the master's degree in computer science and media from the Bauhaus-University Weimar, Germany. He leads the HLRS's strategic development in fields related to artificial intelligence, data analytics, and quantum computing. As a researcher, He has contributed to multiple funded research projects exploring cloud computing technologies, applications of high-performance computing and data analytics to address global challenges, and the development of workflows that integrate AI and high-performance computing.