



Containers in HPC: a survey

Rafael Keller Tesser^{1,2} · Edson Borin¹

Accepted: 21 September 2022 / Published online: 27 October 2022

© The Author(s), under exclusive licence to Springer Science+Business Media, LLC, part of Springer Nature 2022

Abstract

OS-level virtualization (containers) has become a popular alternative to hypervisor-based virtualization. From a system-administration point-of-view, containers enable support for user-defined software stacks, thus freeing users of restrictions imposed by the host's pre-configured software environment. In high performance computing (HPC), containers inspire special interest due to their potentially low overheads on performance. Moreover, they also bring benefits in portability and scientific reproducibility. Despite the potential advantages, the adoption of containers in HPC has been relatively slow, mainly due to specific requirements of the field. These requirements gave rise to various HPC-focused container implementations. Besides unprivileged container execution, they offer different degrees of automation of system-specific optimizations, which are necessary for optimal performance. When we looked into the scientific literature on containers applied to HPC, we were unable to find an up-to-date overview of the state-of-the-art. For this reason, we developed this extensive survey, including 93 carefully selected works. Overall, based on our survey, we argue that issues related to performance overhead are mostly solved. There is, however, a clear trade-off between performance and portability, since optimal performance often depends on host-specific optimizations. A few works propose solutions to mitigate this issue, but there is still room for improvement. Besides, we found surprisingly few works that deal with portability between dedicated HPC systems and public cloud platforms.

Keywords Containers · HPC · High performance computing · Parallel processing · OS-level virtualization · Survey

Edson Borin contributed equally to this work.

✉ Rafael Keller Tesser
rktesser@unicamp.br

Extended author information available on the last page of the article

1 Introduction

A real challenge in *high performance computing* (HPC) environments is to support the wide variety of use-cases and software required by users, without overburdening them or the system administrators. This is even harder because HPC applications commonly have complex dependencies. Moreover, there may be conflicts with other applications, and their own dependencies. This issue is aggravated even more by the growing computational demands from other fields, such as artificial intelligence and big-data, which leads their users to seek the utilization of HPC systems.

We could amend this problem by veering away from the traditional administrator centered software management, and implementing strategies to allow users to configure their own customized software environments. On the other hand, this places most of the burden on the users, and there are also concerns related to system security and stability. Moreover, software portability can still be a challenge, but a worth one, since it reduces user effort, by potentially eliminating the repeated building and configuration effort associated with moving an application to a new system.

Other fields have widely employed *hypervisor-based virtualization* to provide customizable software environments, in the form of *virtual machines* (VMs), while also providing portability and security. HPC, however, has shied away from this approach, due to performance overheads added by the extra layers between applications and hardware. It is worth mentioning that cloud computing instances are most often provided in the form of a VM, especially in public, commercial, clouds. Therefore, this is arguably the most accessible way to run HPC applications in the cloud. In this scenario, the Cloud is a means to provide HPC-capable computing resources for users or institutions that do not have access to, and cannot afford owning, a dedicated HPC system.

A better alternative to hypervisor-based virtualization in HPC is *operating-system level virtualization* (containers). In this approach, the application runs inside a container environment, which is isolated from the host via a combination of OS-kernel level features, such as *namespaces* and *cgroups*. Figure 1 illustrates the main differences between containers and hypervisor-based VMs. As illustrated, an application in a VM has a complete guest OS and a hypervisor between itself and the host OS. Containers, on the other hand, execute directly on top of the host OS, sharing the same OS kernel, and some of the host's software. Thus, containers should provide reduced execution overheads compared to VMs, potentially reaching near native-OS performance. Another advantage is that containers may contain only the software that is actually needed to run the application, thus reducing their size. Compared to VMs, this should make containers faster to deploy, and to transfer between different systems, as well as requiring less storage space.

Containers bring a series of potential benefits to HPC, such as improved software environment flexibility and portability, as well as enhanced research reproducibility. Regarding the first, they should allow users to create their own

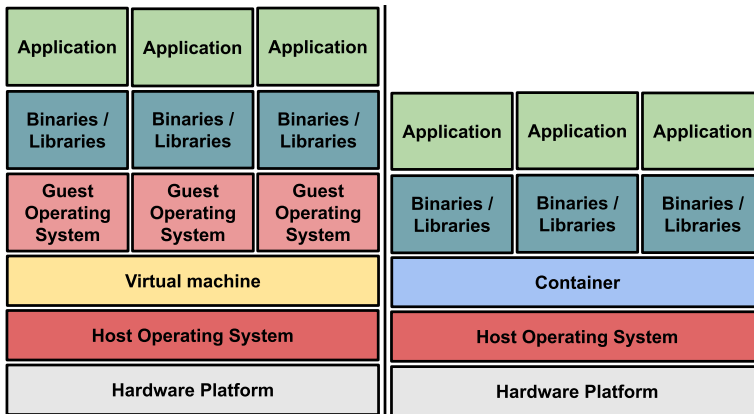


Fig. 1 Hypervisor-based virtualization (VMs, left) and OS-level virtualization (containers, right)

user-defined software stack (UDSS) [69], containing all the necessary software and dependencies. In terms of portability, containers should allow this UDSS to be transferred between different systems, thus reducing the need for software re-installation and reconfiguration, or even eliminating it. In addition, users may employ a base container image as a starting point for their custom environment. To reduce the burden placed on users, developers and sysadmins can provide pre-configured base containers, with commonly used tools, frameworks, and libraries. These images may be either generic and publicly available, or tailored to a particular system.

Reproducibility, in the context of scientific software, refers to the ability of obtaining equal, or sufficiently similar, results when repeating a previous experiment. In computational science, the stability of results may depend on the software itself, on its dependencies, as well as on the whole software and hardware environment. As a direct consequence of their portability, containers should also provide enhanced reproducibility. Building actually reproducible containers, however, entails a careful selection of their contents. Reproducibility may depend on specific versions of software components, such as dependencies (e.g., libraries), and build tools (e.g., compilers).

As the technology advanced, several implementations of containers have been developed, the most prominent being Docker. It was, arguably, the first one to provide a high-level management interface for users, thus being responsible for a huge growth in the popularity of container technology. Earlier container implementations include Linux VServer, and LXC. The latter was the original base layer of Docker.

Despite being considered a *de-facto* standard, Docker never really gained traction in HPC. One reason being that it was designed to support microservices, thus placing a heavy focus on environment isolation. This contrasts with the sharing approach that is most common in HPC. In addition, its containers are launched by a *daemon* that has administrative privileges, which are transferred to any application executed

inside a container. This raises serious concerns regarding the possibility of privilege escalation, especially in HPC systems, which are usually multi-tenant.

Due to these concerns, and as interest in the use of containers in HPC increased, a series of HPC-focused implementations were proposed. Among them, *Shifter* [25], *Singularity* [46], and *Charliecloud* [69] are, arguably, the most popular ones. In order to actually be useful for HPC, a container implementation has to deal with a series of HPC specific concerns. Among the requirements to be satisfied are: (a) Secure implementation, without the risk of privilege escalation. Meaning that containers should not have administrative privileges. (b) Low performance overhead compared to running applications natively on the host. (c) Access to specialized HPC hardware, such as performance accelerators (e.g., GPU, FPGA) and high performance network interconnects (e.g., InfiniBand). (d) Access to specialized software, such as vendor-optimized libraries and tools. Section 3 discusses these concerns in more detail.

In order to advance the state-of-the-art on the use of containers for HPC, it is essential to understand the advancements that were already made, as well as the current state of the research in this subject. With this goal, this survey presents a carefully selected collection of 93 works on this subject, which were published between 2016 and mid-August 2022. To the best of our knowledge, the most recent survey to touch on this topic [15] only includes publications up to 2018. We are also aware of a work published in 2022 by Bentaleb et al. [10], in the form of an overview of container technologies, instead of a survey, but it touches very briefly on HPC, in a short subsection that references only seven other works.

In this survey, we provide some statistics about the collection of selected works and, most importantly, summarize each work's contribution towards overcoming the major challenges related to using containers in HPC. Based on this survey, it seems that the challenges related to performance overheads have already been overcome. The same can be said for the challenge of running containers without administrative privileges. Moreover, their benefits to portability and reproducibility are also confirmed. Conversely, the current strategies to provide access to specialized hardware and vendor-optimized software can hurt container portability. So, users may be faced with a trade-off between performance and portability.

1.1 Organization of this text

The remainder of this text is organized as follows. In Sect. 2 we provide an historical overview of container technology, and present Docker, the *de-facto* standard container implementation in the industry. Next, in Sect. 3 we describe HPC-specific requirements for containers, together with a few desirable characteristics. Following in Sect. 4, we present the main container implementations that were specifically designed for HPC. Then, in Sect. 5, we describe the methodology we employed to search and select works for this survey, and to classify them according to the major challenges they address. In Sect. 6, we present some statistics about the works in this survey, including an overview of the main challenges they address, and of the container implementations they employ. Then we enter into a detailed literature review, starting with Sect. 7, where we describe its organization. This is followed by Sects. 8 to 13, where we summarize

the contributions of each work towards surpassing the main challenges we identified in this survey. These are related to performance overheads (resource-specific and overall), portability, reproducibility, and to running containers without administrative privileges. To close the review, in Sect. 14 we summarize a few related works, which are previous survey or overview papers that are included in our selected publications. Finally, in Sect. 15 we present the insights we derived from the information we compiled, delineate research opportunities, and discuss the future directions of our work.

2 Overview of container technologies

As the technology advanced, diverse container implementations were released to the public. In HPC, however, a series of specific requirements has slowed down the adoption of containers. Moreover, these often clash with the design philosophy of traditional container technologies, especially with their focus in isolation. Nevertheless, there have been many attempts at using containers in this field, which led to the development of HPC-specific container implementations.

Next, we have a historical overview of container technologies. After that, we end this section by presenting Docker [59], the industry's *de-facto* container implementation (Sect. 2.2).

2.1 Historical overview

Early Linux container implementations provided a lower level of abstraction, which restricted their public to users with a deeper system administration knowledge. Launched circa 2005, *Linux VServer*¹ is one of the oldest implementations of *Linux OS-level virtualization*. It uses its own capabilities to achieve isolation, via a *kernel patch*. Isolation of the container's file system is done using *chroot*². Moreover, it employs a global *PID namespace*³ to hide processes that are outside the container, and to prevent inter-container communication between processes. One interesting aspect, especially for distributed applications, is that VServer does not implement network isolation. Therefore, all containers share the host's network subsystems. To avoid containers sniffing communication from or to another container, it adds a filtering mechanism to the network stack. CPU isolation is achieved by overlapping the standard Linux scheduler with a filtering mechanism, to provide fair-sharing and work-conservation of the CPU. Resource limitation is performed using system calls provided by the Linux kernel, and through capabilities added to it by VServer. Moreover, there is support for using *cgroups*⁴ to limit the resource usage of each container. This is helpful in maintaining performance stability when simultaneously executing multiple containers in the same host.

Another early Linux container implementation is *OpenVZ*⁵, which provides similar functionality to Linux VServer but is built on top of *kernel namespaces*. The

¹ <http://linux-vserver.org/> (Acc. in Sept. 2022).

² https://www.gnu.org/software/coreutils/manual/html_node/chroot-invocation.html (Acc. in Sept. 2022).

³ <https://man7.org/linux/man-pages/man7/namespaces.7.html> (Acc. in Sept. 2022).

⁴ <https://www.kernel.org/doc/html/latest/admin-guide/cgroup-v1/cgroups.html> (Acc. in Sept. 2022).

⁵ <https://openvz.org> (Acc. in Sept. 2022).

PID namespace is used to guarantee process isolation between containers. The *IPC namespace* ensures that every container has its own shared memory segments, semaphores, and messages. Finally, it employs the *network namespace*, so that every container has its own network stack (devices, routing tables, firewall rules, etc.). Moreover, a real network device can be assigned to a specific container. OpenVZ also includes a two-level CPU scheduler, which acts at both intercontainer and intra-container levels. Their respective purposes are to ensure fair-sharing of CPU resources between containers, and between processes in the same container.

*Linux Containers*⁶ or simply LXC is one of the most well-known container implementations. As OpenVZ, it employs *kernel namespaces* to accomplish process and resource isolation. Moreover, the *PID*, *IPC* and *network* namespaces are also employed. Besides, it uses *cgroups* to define the configuration of the network namespaces, to limit CPU usage, and to isolate containers and processes. Finally, IO operations are controlled by a *Completely Fair Queuing* (CFQ)⁷ scheduler. Instead of using LXC directly, it is also possible to use LXD⁸, which is a high-level container manager built on top of LXC.

The surge in popularity of containers has really come with the launch of Docker in 2013 [59]. It was initially built on top of LXC, but newer versions employ its own low-level container library. Its user interface provides a higher level of abstraction than LXC, thus being one of its main innovations, and greatly simplifying the creation, building, sharing, deploying, and execution of containers. Docker is considered the *de-facto* container implementation in the industry, being widely employed for the deployment of microservices in the cloud.

On June 2015, the *Open Container Initiative* (OCI)⁹ was launched by Docker, CoreOS and other leaders in the container industry. According to their website, the OCI is “an open governance structure for the express purpose of creating open industry standards around container formats and runtime.” The OCI standards currently include two specifications: the *Runtime Specification* and the *Image Specification*. The *Runtime Specification* defines how to execute a *file system bundle*. According to it, an OCI implementation would download an *OCI image*, which would be unpacked into an *OCI runtime file system bundle*. This bundle would then be run by an *OCI runtime*. The *Image Specification* defines how to create an *OCI Image*, which would generally be done by a build system. The build output would be an image manifest, a file system (layer) serialization, and an image configuration. An OCI image should contain the information required to launch the application on the target platform. To help jumpstart the initiative, Docker donated its container format and runtime, *runC*, to the OCI.

⁶ <https://linuxcontainers.org/> (Acc. in Sept. 2022).

⁷ <https://www.kernel.org/doc/Documentation/block/cfq-iosched.txt> (Acc. in Sept. 2022).

⁸ <https://linuxcontainers.org/lxd/> (Acc. in Sept. 2022).

⁹ <https://opencontainers.org/> (Acc. in Sept. 2022).

2.2 Docker

As mentioned above, Docker[59] is the current *de-facto* standard container technology in the industry. It was designed having in mind the deployment of microservices, with an emphasis on isolation. Besides the runtime system, it provides Docker Hub, a public repository where container images can be stored and shared by common users. Initially, Docker was a high-level user interface for LXC, which later was replaced by its own low-level container library.

Despite its popularity, some problems hamper the adoption of Docker in HPC. First, its focus on isolation goes against the parallel application paradigm of sharing data between processes. One of the main difficulties is to have direct access to specialized HPC hardware, and to the host's vendor-optimized libraries and tools. Both are of high importance in order to achieve the best possible performance from the system.

The second and most important issue is that Docker, by default, requires its daemon to run with administrative privileges. This is usually not an issue in cloud platforms, which employ a security model based on a trusted user running trusted containers. Moreover, in the cloud, the common approach is for the user to have complete control over their instances. Which is possible due to each instance, be it VM or bare-metal, having only one user, and being “*reinstalled*” at every reservation. This way, it should not be possible for one user to modify another user's environment.

Multi-tenant HPC systems, on the other hand, typically have an administrator that manages the system, and users are not allowed to modify it. This way, it is easier to maintain the system security and stability in an environment that is shared by different users. Hence, running a user-defined container launched by a daemon with administrative privileges may introduce security issues.

Despite these difficulties, there were some attempts to use Docker for HPC [4, 8, 57, 64, 68, 71, 75, 81]. Based on these, we gather that the problems of accessing specialized HPC hardware and host optimized libraries are relatively easy to solve. The secure execution of Docker, however, presents a harder challenge. Most of the works employing Docker in HPC simply ignore this issue, with very few works really attempting to solve it. Among these, Sparks [81] implemented Docker plugins to enhance its security, and Azab [4] developed *Socker*, a wrapper to securely run Docker on HPC systems.

In the next two sections, we respectively discuss the HPC-specific requirements for containers, and container implementations that were specifically designed for HPC.

3 HPC-specific requirements for containers

In order to be useful for *high performance computing* applications, container implementations should satisfy the *requirements* enumerated in the next four paragraphs.

Secure implementation: Container technologies must protect the OS against malicious or faulty user software, which is especially important on multi-tenant systems.

Low performance overhead: The performance of HPC workloads inside a container should be close to the one achieved when running the same workload directly on the host.

Access to specialized hardware: HPC systems are usually equipped with specialized hardware, such as computation accelerators and high performance interconnects. Accessing them is essential for getting the best possible performance out of the system. Therefore, containerized HPC applications should be able to access this hardware, with negligible or non-existent performance penalties.

Access to specialized software: In order to get the most performance out of the system, HPC hardware vendors often provide libraries and tools that are optimized to run on their hardware. Therefore, it is essential for containerized HPC applications to have access to this optimized software, which is installed in the host.

3.1 Desirable features for HPC containers

Besides the above requirements, the following features are also *desirable* for HPC containers. The two main ones are presented below.

Enabling the deployment of User-defined Software Stacks (UDSS) [69]: That is, to enable users to create customized environments containing applications and their dependencies. This would enable running their applications independent of the software that is provided in the host. Such independence greatly enhances the portability of the environment, which is important when migrating the application between systems, or when aiming at reproducibility.

Supporting standard container images (OCI/Docker): The use of standard container image formats enhances flexibility and usability. For instance, supporting Docker images would allow the container to be created on the user's personal computer, using Docker, and to be shared via Docker Hub. Moreover, it would be executable both in HPC systems, using an HPC-specific runtime, and in the cloud, using Docker. In addition, this would enable the execution of a wide array of container images already available in public repositories, as well as using these as base images for new containers.

3.2 Possibly unnecessary features

In addition to the above requirements and features, it is important to remark that a few common characteristics of containers are often *unnecessary* for HPC. For instance, for most HPC applications it may not make sense to run more than one container per host. Therefore, *performance isolation between containers may not be required*. Nevertheless, some HPC sites allow multiple jobs to simultaneously share a computer node. In these cases, containers should respect the resource usage limits imposed by the job scheduler. Moreover, in cases where different jobs do not share any computer node, which is common in HPC, there is usually no need for network isolation.

4 HPC container implementations

Based on our survey, the three most popular HPC container implementations are Singularity, Shifter, and Charliecloud. Below, we present a short description of each of these solutions.

Shifter [25, 38] is one of the first HPC container implementations, initially aimed at *Cray* systems. It enables the unprivileged execution of containers, and supports Docker images via conversion to its own image format. So, users can create the images using Docker, thus taking advantage of publicly available Docker images. Shifter integrates an image gateway that serves as a container database and can integrate with Docker Hub. Moreover, it integrates with the Slurm job scheduler, to facilitate the submission of container jobs. In addition, Shifter relies on MPI ABI compatibility to link the container's MPI with the host's, to take advantage of host-specific optimizations and high performance interconnects. Access to GPUs, however, needs to be manually configured. Besides being known to provide near native performance and currently being installable in non-Cray systems, its popularity is hampered by a complex installation process.

Singularity [46] is another HPC-focused container runtime that allows the unprivileged execution of containers. Its installation, however, still requires administrative access. It has its own container image format, but enables pulling and converting Docker images. Singularity provides native support for specialized network interconnects, and for GPU accelerators. Access to vendor optimized software can be achieved by mount-binding into the container a directory from the host, containing the required software. This can be done by passing a specific argument when calling the runtime.

Charliecloud [69] is a runtime that aims to be a lightweight *User Defined Software Stack* (UDSS) implementation. It differentiates itself from the others by having a small code footprint and not requiring administrative privileges, even for installation. Different from Shifter and Singularity, it relies on *User namespaces*, which is an unprivileged namespace that allows users without administrative privileges to create any of the other namespaces. For this reason, Charliecloud requires a new enough Linux kernel to support this feature, which has to be enabled in the kernel configuration. *User namespaces* enable Charliecloud to run Docker containers without administrative privileges. Locally, Charliecloud employs Docker tools but, before execution, the Docker image must be flattened to a single archive file. Since it employs Docker tools that require administrative privileges, this must be done before transferring the image to the HPC system. Charliecloud does not employ any network related namespace, thus it allows direct access to the host's network. Access to devices and file systems is available inside the container by default. User-space drivers, needed for hardware such as GPUs and high performance interconnects, can be used by making the required library files available inside the container. Two ways to accomplish this are installing compatible versions of these inside the container, or bind-mounting a host directory, containing the necessary files, to a directory inside the container. The same approach can be employed to access vendor optimized software.

Table 1 Relevant publication venues considered in the first step of our search**Journals:**

Concurrency and Computation: Practice and Experience

IEEE Transactions on Parallel and Distributed Systems

Journal of Parallel and Distributed Computing

The Journal of Supercomputing

Conferences:

IEEE Int. Parallel & Distributed Processing Symposium (IPDPS)

Int. Conference on High Performance Computing (ISC-HPC)

Int. European Conference on Parallel and Distributed Computing (EuroPar)

The Int. Conf. for High Perf. Computing, Networking, Storage, and Analysis (SC)

Finally, besides the above, a few other container implementations that appear in the works we surveyed are capable of unprivileged execution. Among them are *Podman*¹⁰, *rootless Docker*¹¹, and *udocker* [26].

5 Survey methodology

Our survey methodology is divided in four stages. The first two were the search for papers on the chosen subject, and the refinement of our scope. They are detailed, respectively, in Sects. 5.1 and 5.2. The search process consisted of several steps in which we expanded our list of candidate works based on the results of the previous step. The scope refinement consisted in trimming our list of pre-selected works, by removing those deemed irrelevant or out of the scope of this survey.

In the last two stages, described in Sect. 5.3, we read and classified the selected works. The former consisted of reading the full text of these publications, and registering some data about them. In the latter, we used this data to classify the works according to the main challenges they discuss, and/or attempt to solve.

5.1 Search for relevant works

We started our search by looking for papers on the use of containers for HPC which were published from the year 2016 to 2022. First, we searched for articles on relevant HPC conferences and journals, which are listed in Table 1, using the search terms “*container*” OR “*containers*” OR “*os-level virtualization*”. Since these are HPC conferences and journals, we did not include terms such as *HPC* or *high performance computing*. In our next step, we expanded this search by using *Google Scholar* to look for papers from the same period, but without limiting ourselves to specific venues. We used the search terms (“*container*” OR “*containers*”

¹⁰ <https://podman.io/> (Acc. in Sept. 2022).

¹¹ <https://docs.docker.com/engine/security/rootless/> (Acc. in Sept. 2022).

OR “*os-level virtualization*”) AND (HPC OR “*high performance computing*”). We limited our search to the first 10 pages (100 results), and the results were sorted by relevance (as ranked by the search service). In these two initial stages, we narrowed down our results by scanning the titles of the papers and, if needed, their abstracts. In this process, we were able to eliminate those works that were clearly out of our scope.

The third step in our search was to identify which conferences and journals had most papers listed in the results of the previous stages. We then expanded our results by inspecting all the papers in these venues that matched our search terms and were published in the last considered years (2016–2022).

Originally, our survey only included works published until early 2020. For these, since we had a large amount of results, we did an extra fourth step, where we considered the citations and citation counts, in order to trim-down our results. First we looked into the related works cited by the pre-selected papers to make sure we did not miss any important work. Based on this, we selected to go to the next phase both the works that were frequently cited and those cited in a context that indicated they could be relevant. Second, among the works that were found either via search or via citations, but were not yet selected for the next phase, we selected those that had 100 or more citations on Google Scholar. Later, we updated our survey to include more recent works (up to mid-August 2022), but decided not to do this citation-based step, since the number of pre-selected works was already considerably smaller than in the original 2016–2020 search. At the end of this first stage, we arrived at *185 pre-selected works*.

5.2 Scope refinement

The next stage of our selection process was refining the scope of our search, by selecting only those papers that actually deal with the *use of containers in the scope of high performance computing*. This filtering was done by reading the titles, abstracts, and conclusions of the papers that were pre-selected in our search stage. In some cases, we also looked into their evaluation methodology, to determine if it included any HPC-related metric. During this process, we also gathered statistics for use in a later stage, such as which container technologies were employed, and what aspects of the use of containers were explored by each work.

The first filtering criteria was if the work’s stated proposal was actually related to the use of containers. We selected papers that proposed using containers as part of the solution for a problem, those which evaluated some aspect of their usage (e.g., performance overheads), and those which consisted of literature reviews on the subject. For instance, some papers mentioned containers in their related work, but only evaluated the performance of hypervisor-based virtual machines. Therefore, these were considered out of the scope of this survey.

Next, we created three filtering criteria to define which of these works on containers are in the scope of HPC. The first criterion was if the authors actually stated their work was aimed at solving a problem from the HPC field. The second was if the work evaluated the performance of HPC workloads (benchmarks or applications)

when running inside containers. According to this criterion, the evaluation of performance metrics not specifically related to HPC was not considered enough to qualify the paper as in our scope. The third criterion was if the work was a survey of existing literature that explicitly included works on containers applied to the HPC field. The papers which met any of these criteria were considered to be in the scope of our survey, thus leaving us with a total of *98 selected works*.

5.3 Reading and classification

After the search and selection stage, we read the 98 selected works. Upon reading the full text, it became clear that 5 of these papers did not fit our scope. Therefore, they were removed from the survey, arriving at a total count of 93 papers included in the survey. In this stage, we registered the container implementations being used by each work, and classified them according to the challenges they address. A preliminary classification had already been done during the scope refinement stage. The full reading of the paper allowed us to refine, correct, and expand this classification.

We specifically registered if the work utilized Docker, Singularity, Shifter, Linux VServer, OpenVZ, LXC, or Charliecloud. We were already aware that these were the most popular container implementation, based on the preliminary reading performed at earlier stages. Works that employed other container runtimes or implemented their own, were classified as “*Other*”. We also classified the challenges addressed by each paper into eleven classes: *Communication overhead*, *Storage overhead*, *CPU overhead*, *GPU overhead*, *Memory Overhead*, *Overall application overhead*, *Other Overhead*, *Portability/Migration* (of code or binaries), *Reproducibility*, *Unprivileged container execution*, and *Survey or summary*. The classes for Memory and Overall overheads were late additions, done during the full-reading stage. In the case of research surveys, we registered the implementations that were discussed in the text, and these works were classified as belonging, exclusively, to the *Survey* class. In Sects. 8 to 13, we summarize the contributions of the selected works towards understanding and surmounting the challenges above.

6 Statistics

In this section, we present and analyze a few statistics about the papers included in this survey. This should help to understand aspects like the challenges researchers are trying to solve by using containers, and the popularity of different container implementations in the scope of HPC.

First, we present the distribution of the number of papers by year of publication. Next, we take a look at the total number of papers that employ each container technology, to which we add an analysis of how their usage evolved over the years comprehended in our survey. Finally, we perform the same analysis for the number of papers that address each of the challenges included in our classification.

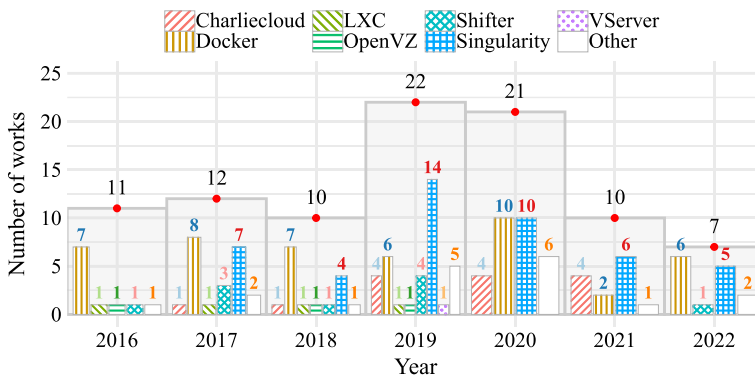


Fig. 2 Total number of works by year of publication (gray area, red points), and number of works that employed each container implementation on a specific year (color figure online)

6.1 Year of publication

The shaded area in Fig. 2 represents the number of selected works per year of publication. We can observe a small increase in publications from 2016 to 2017. This coincides with the appearance of HPC-focused container runtimes, like Shifter and Singularity. In 2018, however, it seems the research interest in the subject has decreased, but not by much. Nevertheless, going from 2018 to 2019, the number of publications more than doubled. This coincides with a large increase in popularity for Singularity, and also a small increase in the interest on Charliecloud and Shifter. For 2020, the number of works remained almost the same, thus indicating a continued interest in the subject. In 2021 and 2022, however, we see a considerable decrease in the number of publications. One reason could be that most issues regarding containers in HPC, especially the ones related to performance, already had well known solutions by then. Moreover, we were only able to include works published until mid-August 2022, which was when we finished our search.

6.2 Employed container technologies

The colored bars in Fig. 2 represent the number of selected works that analyzed each container implementation, by year of publication. Initially, in 2016, the only higher level implementation to show up is Docker. We also have the first appearance of an HPC container implementation, Shifter. In addition, we have a few studies employing lower level implementations, such as OpenVZ, Linux VServer and LXC.

Along with a significant increase in the number of works, 2017 marks the first appearance of Singularity, already close to Docker in number of works. Charliecloud is another implementation that appears for the first time in 2017. Moreover, Shifter shows some growth but is not close to the number of papers that deal with Singularity containers. The “other” category also shows some growth, due to attempts by individual research teams to craft their own container solution for HPC. In 2018,

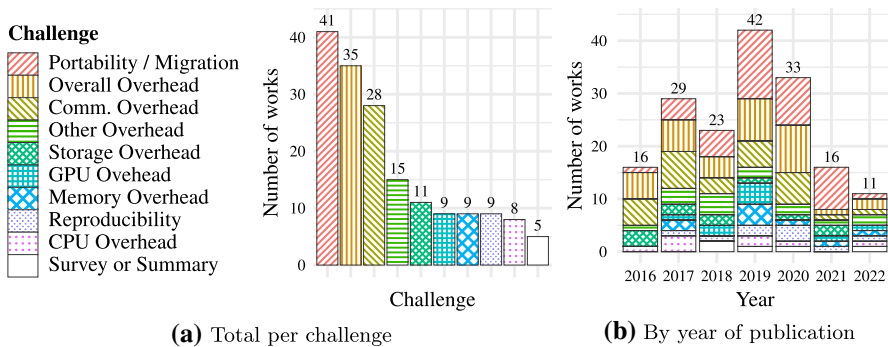


Fig. 3 Challenges that were addressed by the selected works

all implementations, except Charliecloud, show a decrease in the number of papers, following the previously mentioned decrease in the total number of published works on containers for HPC. Docker remains in the lead, with Singularity in the second place. The significant increase in the number of publications in 2019, even surpassing 2017, seems to be a result of the growing interest in Singularity, which took the lead from Docker, by a wide margin. The number of works on Docker containers, on the other hand, remained stable from 2018 to 2019.

In 2020, Singularity remained in the lead, and there was a small increase in the number of publications dealing with Docker containers. Charliecloud maintained an almost stable publication count and, for the first time, there were no publications employing lower level container implementations. Interestingly, for this year we could not identify any paper using Shifter inside our scope. Singularity remained in the lead in 2021, while Docker and “Other” showed up less frequently, and Charliecloud took second place, but remained at the same level as in the two previous years. Interestingly, in 2022 Docker returned to the first place, even if it surpassed Singularity by only one work.

6.3 Addressed challenges

Figure 3a shows the total number of selected papers that address each challenge from our classification (Sect. 5.3). The first thing we notice is that the main challenge being attacked is *portability*. Which is not surprising, since that is one of the main advantages containerization brings to HPC. Following it closely, we have the concern with the *overall application execution overhead*. That is, the potential decrease in performance of the application as a whole, without taking into account specific components of this overhead (such as, computation, memory, and communication overheads). Other works evaluate resource-specific overheads. Among them, the *communication overhead*, which takes the third place in our chart, and represents one of the main concerns when containerizing parallel applications. Below, but still standing out, we have the storage overhead. Continuing with the resource-specific performance, we have a few works that measured pure CPU overhead, the overhead for GPU accelerated computation, and others that measured the overhead

for memory operations. The “*other overheads*” category includes different types of overhead that did not fit into the other categories (e.g., memory consumption, energy consumption, deployment time, and startup time). Therefore, the fact that it includes more works than some resource-specific overheads is not a concern. Surprisingly, it seems that few works focus on *reproducibility of research*. Only the *survey or summary* category has fewer works, four being presented as literature surveys, and one as general overview, or summary, about container technology.

Figure 3b shows the evolution of the interest on each challenge along the years. For 2016, most works analyzed the communication overhead of containers, as well as their overall overhead on HPC applications. This trend continues in 2017, for which we can also observe a significant increase in works related to portability and to the overall overhead. While the interest in portability remained in 2018, the number of works analyzing communication overheads decreased. This could indicate a loss of interest in this subject, due to the appearance of technologies that solve this issue, such as HPC-specific container implementations, and of mechanisms that enable Docker to directly access HPC network hardware. In addition, the interest in the overall performance overhead for applications did not decrease as much as in the other overheads. There was also a small increase in works analyzing overheads we did not include in our classification. In 2019, we see a large increase in works dealing with most challenges, except storage overhead. Notice the large increase in the number of works concerned with portability. Looking back at Fig. 2, we could attribute most of this growth to the increased popularity of Singularity. 2020 presented a similar distribution of concerns as the previous year, but there were fewer works dealing with portability, memory and CPU computation overheads, and no works addressing GPU overheads. In 2021, most works were still concerned with portability, while interest in the other challenges decreased. This may be due to most of them already having well known solutions by then, especially the performance-related ones. Moreover, based on our survey, many solutions for preserving performance inside containers tend to compromise portability. Finally, in 2022 we see a decrease in papers on portability, but this data is still incomplete as we finished our search in mid-August 2022.

7 Organization of this survey

In Sects. 8 to 13, based on our classification, we summarize how the selected works deal with each of our pre-defined challenges. Many of them evaluate some type of performance overhead that may be caused by containerization. Such works compare the performance of different container technologies to the performance of bare-metal, of other container implementations, or of hypervisor-based virtualization.

Some experiments measure the performance of specific operations or system resources (hardware or software), to evaluate *resource specific overheads*. These are related to communication, permanent storage, CPU computation, GPU computation, and volatile memory. A last subclass, called *other overhead*, includes experiments that do not fit in the previous ones. For example, those that evaluate the startup overhead caused by container deployment.

Other experiments evaluate the performance overhead of applications as a whole, without paying attention to specific operations or system resources. Such applications can be full-fledged real-world applications, benchmark applications, or even benchmark suites that include several benchmark programs (e.g., NPB, HPCC). According to our classification, these experiments evaluate *overall application overheads*.

The experiments on resource-specific and overall application overheads are, respectively, in Sects. 8 and 9. In addition, Sect. 10 includes overhead-related experiments that do not fit into our pre-defined categories. Next, in Sect. 11, we move to the challenge of application portability, and in Sect. 12 we deal with reproducibility. Finally, in Sect. 13, we present works that deal with the execution of containers without administrative privileges.

We feel necessary to reinforce the fact that we classified each experiment according to its goals. That is, the purpose the researchers expected it to fulfill. By focusing on goals, instead of tools (software and hardware), it becomes easier to match the experiments to our pre-defined challenges. For this reason, a specific application or benchmark may appear in different sections, since different works may have employed it with different objectives. For example, one work may have used it to compare containerized and bare-metal performances, while another may have evaluated how different network configurations impact its performance. We would consider these, respectively, an *overall application overhead* experiment, and a *communication overhead* experiment.

8 Resource specific overheads

In this section, we present experiments aimed at evaluating different kinds of *resource specific overhead*, which are overheads in operations that utilize specific hardware or software resources. This kind of exploration normally employs benchmarks that stress that resource, to discover potential performance limitations, or to compare different environment configurations. We classified these experiments according to types of overhead, each related to a specific system resource. These include overheads in communication (shared memory and network interconnection), computation (CPU and GPU), permanent storage (local, remote/networked, and parallel file systems), and volatile-memory operations. Next, we dedicate a subsection to each of these classes.

8.1 Communication overhead

Chung et al. [20] evaluated the communication performance of Docker over InfiniBand, using tests from the IB driver and the *OSU Micro-benchmarks*. In both, Docker's point-to-point performance was equivalent to the physical machines. Moreover, Docker performed better than VM in the OSU collective communication benchmarks.

Bahls [6] employed IMB to evaluate the network performance of Shifter on 128 nodes of a Cray system. The containers were configured to rely on MPICH-ABI compatibility with the host's Cray MPI to access the high performance interconnection. They compared the performance of containers with two different versions of IMB, one compiled with an open-source compiler, and another built with the Cray Compiling Environment (CCE). Many MPI routines achieved near-native performance inside the containers, but several others, including `Gather`, `Reduce_scatter`, and `Scatter`, were up to 3 times slower than the native CCE-compiled version.

Herbein et al. [30] proposed a series of techniques to improve resource allocation for HPC application containers. Among these, they implemented a mechanism that enables setting bandwidth limits for containers (throttling) and also provides a preferential delivery service to critical application containers (prioritization). Moreover, it can "*control latency and delay while providing a way to reduce data losses.*" Tests demonstrated that their prioritization implementation works, and that containers with equal priority receive an equal share of the available bandwidth. The throttling functionality was evaluated by executing three containers with different bandwidth limits in the same node. The results show that their implementation of container network throttling efficiently enforces bandwidth limits.

Traditional MPI implementations employ their *network channel* for communication between co-resident containers, and the much faster *Shared Memory* (SHM) channel or *Cross Memory Attach* (CMA) channels for communication inside a container. Zhang et al. [96] proposed a locality-aware MPI library for container-based HPC clouds, which dynamically detects co-resident containers, enabling them to communicate through SHM or CMA. They demonstrated that, even with a constant process count, the performance of inter-container communication through the network channel decreases as the number of co-resident containers increases. To evaluate their container-locality aware MPI, they used Docker containers to run a series of benchmarks and applications on bare-metal hosts of the Chameleon Cloud. They used the OSU Micro-benchmarks to measure MPI communication performance between two co-resident containers. Compared to default MPI, two-side point-to-point communication performance improved by 79% in latency, and 191% in bandwidth, and bidirectional bandwidth improved by 407%. One-side point-to-point communication showed improvements up to 95% in latency and increased the bandwidth by a factor of 9. The performance of collective MPI communication for 64 containers across 16 nodes (256 processes total) improved from 28%, for `MPI_Alltoall`, up to 86% for `MPI_Allgather`. All these experiments achieved significantly low overheads (up to 9%) compared to the native environment. The authors also evaluated the performance of Graph'500 and NPB (Class D). Their MPI implementation also improved the performances of both Graph'500 (up to 16%) and NPB class D (up to 11% for CG), with small overheads (respectively 5% and 9%). Thus, it was able to remove the bottleneck in the intra-node inter-container communication.

Zhang et al. [95] evaluated the performance of Docker containers using *PCI Passthrough* (Container-PT) and KVM VMs using either *PCI Passthrough* (VM-PT) or *SR-IOV* (VM-SR-IOV) to directly access the InfiniBand hardware. They

tested IB verbs, MPI communications (OSU Micro-benchmarks), and applications (Graph'500, SPEC, NPB, and LAMMPS) in bare-metal nodes of the Chameleon Cloud, equipped with a 56 Gb/s InfiniBand adapter. For both the IB-level and the MPI-level *point-to-point communication* experiments, VM-PT displayed lower latencies than VM-SR-IOV, while Container-PT presented larger latency than VM-PT. The point-to-point throughput of all three configurations was similar to native. In the MPI-level *collective communication* benchmarks, Container-PT achieved the smallest overheads compared to native executions ($\approx 10\%$). Finally, application performance with Container-PT was near-native, and better than with both VM configurations. The authors argue that, even if SR-IOV resulted in worse performances for VMs, its ability to share IO devices between multiple virtual environments provides the necessary flexibility for building large-scale HPC clouds.

In a work that proposed a container-based HPC ecosystem for OpenPOWER systems, Kuity and Peddoju [43] ran the HPCC benchmarks on a single-socket IBM Turismo SCM POWER8 machine (3.857GHz, 8 cores \times 8 threads, 128 GB RAM, 10 Gb/s Ethernet). They observed the median of 20 results of each benchmark, for Docker, KVM, and bare-metal. Network performance was measured with the HPCC-included *PTRANS* (parallel matrix transpose) and *b_eff* (MPI point-to-point latency and bandwidth measurements) benchmarks. Their results from *PTRANS* (50,000 \times 50,000 processes, with 2×16 , 3×16 and 4×13 process grid sizes), show no bandwidth overhead for containers when running 48 or 56 threads. With 8 and 16 threads the containers presented significantly higher median bandwidths than bare-metal, and with 32 threads it was slightly higher than bare-metal. VM was the slowest configuration, except for 16 threads, for which it presented the highest bandwidth, being slightly higher than the one for containers. No explanation was provided for these differences in behavior between different thread counts. In addition, there are a few issues with the presentation of the results. First, it is not clear, however, what these thread counts represent. Second, the authors say that performance degrades when they increase the number of threads, but their graphics show the reverse. For *b_eff* (8byte messages), the latency results displayed the same behavior for the container and the bare-metal ecosystem, and higher latency for the VM. Similarly, the measured bandwidths of container and bare-metal were nearly identical.

As part of the evaluation of their *Virtual Container Cluster* (VCC) model, Higgins et al. [31] employed *HPCC* to evaluate its MPI communication performance. Their Docker-based solution uses *Weave* to create a *Software Defined Network* (SDN). On two 4-node testbeds equipped with Gigabit Ethernet, it achieved 5-7% overhead in the *random ring bandwidth* benchmark, compared to bare-metal. On a 16-node testbed equipped with InfiniBand, however, the results were much worse, with bandwidths 42-45% slower than bare-metal. In the *random latency* benchmark, they observed 20-30% overhead over bare-metal. Despite these results, the authors considered that “the Weave VXLAN overlay network has good scalability attributes and a predictable performance response, in the context of a parallel communication network”.

Zhang et al. [97] evaluated the MPI communication overhead of Singularity, by running the OSU Micro-benchmark on 32 bare-metal nodes of the Chameleon

Cloud (2×12 -core Intel Xeon E5-2670 v3, 128 GB RAM, 56 Gb/s InfiniBand) and on four *Intel Knights Landing* (KNL) nodes (Intel Xeon Phi 7250 @ 1.4GHz, 96 GB RAM, 16 GB MCDRAM, *Intel OmniPath* interconnection). They detected a minor performance overhead for internode point-to-point communication in the Chameleon Cloud nodes ($<7\%$). In the KNL platform using the in-cache memory mode, they detected higher intra-node latencies, and internode latency overheads remained below 8%. For KNL in flat memory mode, Singularity was able to deliver near-native performance. The MPI collective communication performance was evaluated with 52 processes across the 32 Xeon-E5 nodes, and 128 processes across 2 KNL nodes. Overall, singularity achieved collective communications overheads smaller than 8%. Moreover, the authors highlighted the substantial benefits of KNL's flat memory mode for collective operations with messages larger than 256 kB (16–67%). Singularity was able to consistently reflect this behavior.

Kovacs [42] used iPerf to compare the network throughput of LXC, Docker, and Singularity between two hosts, in comparison with KVM and native execution. With their default configuration (bridged networking), LXC and Docker fell behind both native and KVM execution. They also detected high retransmission rates, for both LXC and Docker, when using bridged networking. With host networking, LXC, Docker, and Singularity (host networking by default), achieved similar performances to the native environment.

Le and Paz [48] evaluated the network performance of Singularity on the *San Diego Supercomputer Center* (SDSC) *Comet* supercomputer. The *IMB Sendrecv* benchmark demonstrated slightly lower than native bandwidth for Singularity. Network latency measurements by the *IMB Ping-pong* benchmark showed interleaved latency results between containerized and native executions. The OSU benchmarks displayed slightly different results, with the non-containerized run presenting a slight increase in latency for message sizes above 400,000 bytes. Overall, this evaluation shows that Singularity incurs very small overheads for MPI communication.

Also using the OSU Micro-benchmarks, Arango et al. [3] evaluated the network performance of LXC, Docker and Singularity containers. All three of them showed better latencies than bare-metal. The smallest latency was achieved with LXC ($> 32\%$ of bare-metal), while Singularity's was slightly better than Docker's, which was slightly better than bare-metal's. Their measurements also show a significant advantage for LXC in terms of bandwidth, compared to bare-metal, while Docker and Singularity presented significant overheads. While we cannot be sure, since this information is not clearly presented in the paper, these results may be explained by the lack of a high performance interconnection.

The work by Younge et al. [93] includes the evaluation of the communication overhead of containers by executing the same IMB container image on a *Cray XC30* system and on Amazon Web Services (AWS) *EC2*. On the *Cray* system, Singularity with Cray MPI achieved near native bandwidth, while with Intel MPI the performance dropped to 39.5% of the native bandwidth. On Amazon EC2, the measured bandwidth was only 6.9% of the peak bandwidth achieved through the Cray's *Ares* network. In the native Cray environment, the ping-pong benchmark displayed an 11.3% latency difference between running IMB with static or with dynamically linked libraries. This is an important result, because the containers employ dynamic

linking of the host's optimized libraries. The ping-pong latency in the Singularity container with Cray MPI and of the dynamically linked native executions presented no statistically significant difference. On the other hand, using Intel MPI resulted in a large negative impact, with latencies over 6000 times the ones obtained with Cray MPI. On EC2, the latencies were even longer than for Intel MPI on the Cray system. The `MPI_Allreduce` benchmark measured the latency across 768 MPI ranks (32 nodes on the HPC system, 48 nodes on EC2). It showed a small overhead when using dynamically linked Cray MPI natively, and no difference in performance compared to the containers.

Chen et al. [19] measured the network performance of different back ends of their *Build and Execution Environment* (BEE), when transferring data between two processes. *BEE-VM* (Docker over KVM) and *BEE-Openstack* (Docker over bare-metal Openstack cloud nodes) used InfiniBand via SR-IOV. *BEE-Charliecloud* (Charliecloud directly on top of the native environment) and *BEE-AWS* (Docker inside a Xen VMs, i.e., Amazon EC2 instances) used Ethernet. Hpcbench, with message sizes from 1 to 2^{22} bytes, showed that all four back ends can provide similar network bandwidth and latency to the baseline, for both *point-to-point* and *all-to-all* communication.

In a case study on the use of containers in the *Blue Waters* supercomputer, Belkin et al. [7] evaluated the communication overhead of MPI applications inside Shifter containers. They ran the `MPI_Bcast`, `MPI_Reduce`, `MPI_Alltoall`, and `MPI_Alltoallv` benchmarks from the OSU Micro-benchmarks on 4 to 64 nodes (64 to 1024 ranks). Their results suggest that the performances in Shifter and in the native *Cray Linux Environment* (CLE) are statistically the same.

Yet another work to employ the OSU Micro-benchmarks was published by Saha et al. [75]. They evaluated the latency of different networking configurations for running containers. The benchmarks ran on six 48-core nodes of Chameleon Cloud, interconnected via InfiniBand. RDMA communication with the InfiniBand hardware was enabled in the containers by mapping the host's drivers into them. They tested three Docker container setups, and one Singularity setup. The Docker setups consisted of one container per node with host networking, one container per node with overlay network (via Docker Swarm), and multiple containers per node using an overlay network (varying the number of MPI ranks per container). The *OSU Alltoallv* collective communication benchmark with 65,536-byte messages did not display any significant change in latency for the approaches with a single container per node. Moreover, executing multiple MPI processes per container resulted in a maximum increase of 0.82% in latency, when running one MPI process per container, and 0.72% with 4 processes per container. In another set of experiments, Saha et al. [75] measured the container networking overhead of Ethernet compared to InfiniBand. Once again, they employed six 48-core bare-metal nodes of the Chameleon Cloud. The OSU benchmark measured an increase of $\approx 245\%$ in network latency when using Ethernet, compared to InfiniBand. For MiniFE, a CPU-intensive benchmark, the Ethernet performance overhead was 1% for bare-metal, and 1.4% with Docker. Meaning, the CPU performance hit from using a slower network was minimal. For KMI Hash, a memory-intensive benchmark, the performance degradation of using Ethernet was 84% on bare-metal and 85% on Docker, when compared

to InfiniBand on bare-metal. This demonstrates the importance of having access to specialized high-bandwidth and low-latency interconnects, especially for memory- or network-intensive applications.

Sande Veiga et al. [77] also employed the *OSU* benchmark. This time, to measure the containerized bandwidth of InfiniBand in the *FinisTerra II* cluster, with three different MPI versions. They only executed point-to-point tests, which resulted in latency values which were “correct values for InfiniBand networks.” Moreover, the bandwidth reached ≈ 6 GB/s which is “a correct value range for InfiniBand.”

Hu et al. [35] employed the *OSU* Micro-benchmarks to test the MPI-level network bandwidth overhead of Singularity over 1 Gb/s Ethernet. For message size 1 GB, Singularity’s bandwidth was 1.87% smaller than native, while the average bandwidth for other sizes was 3.72% larger than native.

Continuing the work presented on Saha et al. [75], Beltre et al. [8] evaluated the usage of container orchestrators to run HPC workloads in the cloud. They measured the TCP/IP and InfiniBand network overheads of employing Docker Swarm and Kubernetes, compared with bare-metal. For this purpose, they ran the *All-toallv* (latency) and *OSU Bi-directional* (bandwidth) benchmarks over six 48-core nodes of Chameleon Cloud. The TCP/IP results showed substantial latency and bandwidth overheads for both orchestrators. Kubernetes, whose bandwidth stagnated for message sizes larger than 4 kB, presented the worst results. With Docker Swarm, however, network performance gets closer to bare-metal as message sizes increase. The InfiniBand experiments showed much better results, with both solutions attaining only $\approx 2\%$ average overheads relative to bare-metal.

Maliszewski et al. [54] proposed using *network interface* (NIC) *aggregation* to improve the performance of HPC applications in container-based cloud platforms. To evaluate this approach, they aggregated 1 to 4 NICs as slaves of a virtual NIC, using network bonding (balance round-robin mode). They used network bridging to provide access to the aggregated interface from inside LXD containers, on an *OpenNebula* managed cloud. Network evaluation with *NetPIPE* showed significant latency improvements for both LXD and native executions. Moreover, LXD using 4 aggregated NICs overcame the native throughput at the 10 kB message size. The authors also evaluated the performance of the NAS Parallel Benchmarks on two nodes, each running 16 MPI processes. Overall, LXD achieved higher performance, with up to 38.95% improvement for a network intensive application (IS). The native version suffered small performance losses when aggregating 3 and 4 NICs. The authors assign this to an excessive retransmission of packets by the networking implementation of the underlying OS.

Continuing the previous work, Maliszewski et al. [55], performed new experiments, now using *bonding mode 4* (IEEE 802.3ad), with three hash policies (layer 2, layer 2+3, and layer 3+4), aggregating either 2 or 4 NICs. Their results demonstrated $\approx 40\%$ improvement for IS, with 4 aggregated NICs. BT and SP presented only small performance improvements, which the authors attribute to the use of blocking communication. Except for FT, the layer 2 policy resulted in performance improvements, while the other two caused performance losses. For FT, the higher layer hash policies actually improved performance, which the authors say is due to its intensive network usage.

On experiments with *Alya*, a *computational fluid-dynamics* (CFD) code, Rudyy et al. [72] evaluated the performance of container images executed on an HPC cluster and on the MareNostrum supercomputer, using Singularity, Shifter, and Docker. On the cluster, they identified a slowdown when running on Docker, in the section of the code that performs MPI communication. Shifter and Singularity did not present such slowdown, and achieved similar performance to bare-metal. *Alya*'s *FSI* code (*fluid structure artery simulation*) was tested on MareNostrum, using both a Singularity image with access to system resources (system-specific), and another without it (self-contained). While the system-specific container performed similar to bare-metal, the self-contained one suffered from performance degradation in the same part of the code as in the previous experiments with Docker. This reinforces the importance of containerized HPC applications having access to host resources, such as high performance fabrics.

Lim et al. [50] evaluated five Docker network configurations, being *Docker Linux Overlay*, *Weave Net*, *Flannel*, *Calico with IP-in-IP*, and *Calico without IP-in-IP*. Based on the results of small-scale experiments (2 to 4 nodes with 10 Gb/s Ethernet) using iPerf3, Hpcbench, OSU Micro-benchmarks, and HPL, they proposed Calico as the best performing solution, being comparable with bare-metal. For the OSU and HPL benchmarks, they also tested a Singularity container. Docker with Calico, and Singularity achieved similar levels of performance to bare-metal, for both benchmarks.

Peiro Conde [67] used the OSU Micro-benchmarks to evaluate Singularity containers on an ARM-based cluster with an InfiniBand interconnection. Overall, Singularity achieved higher bandwidth than bare-metal for small messages, albeit by a small margin. With larger message sizes (from 2^{10} to 2^{20} bytes), the bandwidth of Singularity and bare-metal behaved similarly. Moreover, for message sizes between 2^{10} and 2^{15} bytes, Singularity performed significantly better than bare-metal. The authors attributed this behavior to differences between container and host libraries.

Simchev and Atanassov [78] evaluated the performance of two Kubernetes-based MPI clusters: one with all containers mapped to different physical hosts, and another with every 2 containers mapped to the same physical host. These were deployed on four cloud instances with 2 virtual CPUs and 4 GB of RAM each. Through experiments employing the Intel MPI Benchmarks they found that the configuration with shared physical hosts obtained better MPI-communication performance. They also concluded that "*the overall performance of MPI operations is mainly determined by the networking implementation.*"

In a work aimed at analyzing the portability and usability of user-defined containerized stacks, Ruhela et al. [73] evaluated the performance of Singularity, Charliecloud, and Podman, at various problem scales. To evaluate their communication overhead, they ran the Intel MPI Benchmarks on nodes of the Frontera supercomputer. Their MPI_Allreduce experiments on 4000 nodes show nearly the same latency for the native execution and for the two container solutions. The authors remark that, in order to achieve performance at scale for containerized HPC, one must afford the cost of building support for high performance interconnects.

Liu and Guitart [51] analyzed the performance of different container deployment schemes for HPC workloads. They evaluated different container granularities, as well as the benefits of setting memory and CPU affinity for the containers. Their experiments compared Docker using bridge networking and three Singularity configurations, (1) with host-networking and no cgroups (default), (2) with host-networking and using cgroups for resource isolation, and (3) with bridge networking and cgroups. They executed applications from the HPCC benchmark suite (DGEMM, FFT, STREAM, PTRANS, RandomAccess, and b_eff) on a single dual-socket host with 18 CPU cores, of which they only used 16. Results for the two configurations that use bridge-networking, show the expected performance degradation for MPI communication on multi-container workloads. In this configuration, the authors explain, processes on different containers can not detect if they reside in the same host. Thus, they will not attempt shared memory communication. By default, Singularity is not affected by this issue, since it employs host-networking. In addition, over-subscription (32 processes on 8 CPU cores) resulted in some imbalance, which was remedied by setting-up CPU affinity to pin processes to specific cores. Moreover, setting *memory affinity* improved the performance of distributed memory benchmarks in NUMA (*Non-Uniform Memory Access*) platforms. Finally, the authors did not observe any performance variation related to a specific container technology or granularity.

To evaluate the communication overhead of containers at large scale, Ruhela et al. [74] ran the IMB MPI_Bcast and MPI_Alltoall benchmarks. MPI_Bcast was run in a cluster equipped with 20 Gb/s InfiniBand, with dual-socket *Intel Xeon Platinum 8280* (56 cores per node). Results with both 1 and 56 processes per node (one process per container) on a total 6144 nodes showed on par communication latency by Charliecloud and bare-metal. The MPI_Alltoall benchmarks ran on an InfiniBand-equipped cluster with 3.2Tbps peak bandwidth, and 256-core nodes (dual-socket *AMD EPYC 7742*). The results showed similar communication behavior for Charliecloud and bare-metal, with negligible differences in latency.

Liu and Guitart [52] evaluated the performance implications of running multiple containers per node instead of a single one. They evaluated host-networking and overlay networking configurations, but multi-container per host scenarios were only tested with overlay networking. The overlay configurations included TCP/IP (TCP/IP over Ethernet), IPoIB (TCP/IP over InfiniBand), and RDMA (direct access to the InfiniBand hardware via RDMA). Single container tests used Docker and default Singularity, while multi-container used Docker and Singularity-instance (containers running in the background as isolated instances). With a single container per host, they ran the *OSU MPI_Alltoallv* benchmark with 128 processes on 4 hosts and the *OSU Bidirectional Bandwidth* benchmark. Their results showed that RDMA is faster than IPoIB, which in turn is faster than TCP/IP. Moreover, all configurations that did not rely on overlay networking had the same performance as bare-metal, while overlay networking caused large latency and bandwidth degradation. On a second set of tests, they ran the MPI_Alltoall benchmark while continually increasing the number of containers until each had a single process. RDMA presented the best latencies, followed by IPoIB, and then by TCP/IP. Moreover, all configurations presented substantial increase in latency as message sizes increased, especially for

medium (256 B–16 kB) and large messages (32 kB–1 MB). In addition, there were large differences in latency between different network configurations. For the largest message size (1 MB), RDMA achieved $\approx 400 \times 10^3 \mu\text{s}$, IPoIB $\approx 2.5 \times 10^6 \mu\text{s}$, and TCP/IP $\approx 30 \times 10^6 \mu\text{s}$. On a third experiment, to evaluate MPI communication intensive workloads, they ran the HPCC's RandomRing, G-PTRANS, G-FFT, and G-RandomAccess benchmarks. When increasing the number of containers, the first three presented similar performance on TCP/IP and IPoIB and degraded performance for RDMA, while G-FFT presented degrading performance on all overlay configurations, but more pronounced on RDMA. This was explained by network communication being the bottleneck for TCP/IP and IPoIB, while for RDMA the bottleneck was memory performance. In a fourth battery of tests, they ran two MPI throughput benchmarks, namely EP-STREAM (memory bandwidth) and EP-DGEMM (computation). EP-STREAM performed similarly with all container counts, while EP-DGEMM showed improvements with one container per process when using cgroups (to restrict them to specific cores). Both benchmarks demonstrated penalties due to overlay networking. The last experiment by Liu and Guitart [52] evaluated the impact of CPU and memory affinity on multi-container deployments. Overall, enabling affinity did not solve any of the performance degradation in the communication-intensive benchmarks. Nevertheless, EP-DGEMM on all network configurations and G-FFT on RDMA had significant performance improvements. Moreover, the multi- and single-container performances of EP-DGEMM became on par. Finally, the performance of G-RandomAccess on IPoIB degraded, due to load imbalance between processes, but had a small improvement on RDMA, which is more sensitive to memory latency.

Overall, based on the experiments summarized in this section, we can conclude that *containers are capable of providing near-native network communication performance*. For this purpose, however, they must be configured to provide direct access to the network hardware from inside the container. The works discussed above demonstrate and evaluate various solutions to achieve this goal. In most cases, network isolation is sacrificed in exchange for performance. This should not be a problem, since that is not a requirement for HPC. In fact, dedicated HPC systems usually allow applications to run without any network isolation. One aspect that needs some attention, however, is that some solutions require having a copy of system specific libraries inside the container. Therefore, one must consider the consequences for portability when choosing whether to employ such approach (more details in Sect. 11).

8.2 CPU computation overhead

In their work on improving resource management for HPC applications running in containers, Herbein et al. [30] introduced a CPU allocation mechanism that allows Docker to define a dynamic time-slice for each container. Containers are allocated CPU cores according to a round-robin police, and each container is allowed exclusive usage of the core until their time-slice ends. This mitigates performance losses due to frequent context switches and improves cache efficiency. In experiments

where an independent instance of *LINPACK* was executed in each container, the time to run 5 containers decreased from ≈ 20 times the time to run a single container to ≈ 5 times. It is worth mentioning that the length of the time-slice had to be carefully tuned in order to find its optimal value for the target application and system.

Kuity and Peddoju [43] evaluated the computational performance of a system composed of single-socket OpenPOWER machines, equipped with an IBM Turismo SCM POWER8 CPU (3.857GHz, 8 cores \times 8 threads, 64 kB L1 data cache, 32 kB L1 instruction cache, 512 kB L2, 8 MB L3), 128 GB RAM, and 10 Gb/s Ethernet. They ran the HPL, DGEMM, and FFT benchmarks (part of HPCC) in CHPCE containers (their own HPC container runtime), KVM, and on bare-metal. Based on 20 executions, HPL was slightly faster in the container, and 2% slower in the VM, compared to bare-metal. For DGEMM, they ran both *SingleDGEMM* and *StarDGEMM*. The former initially presented slightly better than native performance, but it proportionally dropped as the number of threads increased. Nevertheless, it decreased at a slower rate than the VM executions. *StarDGEMM* also displayed this performance degradation, with containers eventually becoming even slower than the VM. For FFT, *StarFFT* performed better than *G-FFT*, but still shows performance degradation as thread count is increased. The authors attribute the displayed performance degradation to an increase in memory contention.

Aiming to evaluate their CPU computation overheads, Kovacs [42] ran *sysbench* inside LXC, Docker, Singularity, and KVM virtual environments. On a single 16-core server, all three container solutions achieved native performance, while the VM-based executions were slightly slower.

Arango et al. [3] employed HPL-LAPACK to measure the CPU overhead of LXC, Docker, and Singularity. Singularity achieved the best result, with an average performance 5.42% better than native. Compared to native CPU throughput, Docker was 2.89% slower, and LXC was 7.76% slower.

Hu et al. [35] employed *HPL* to evaluate the CPU overhead of Singularity. It achieved near native performance (only 0.4% average overhead) on a 5-node parallel execution.

As part of an evaluation of Charliecloud, Shifter, and Singularity, Torrez et al. [84] measured their CPU overheads compared to bare-metal. All four environments achieved nearly identical performances for *sysbench*, using 36 threads in a single node (3 CPU sockets \times 12 cores).

Gantikow et al. [24] analyzed the suitability for HPC of the Podman runtime, which enables running unprivileged containers. Among others, their CPU performance measurements with *sysbench* did not show any significant computation overhead from containerization ($\leq 0.4\%$). Based on this, and on application overheads (see Sect. 9), they concluded that, besides having a different focus, Podman is suitable for HPC.

Liu and Guitart [52] measured the impact of running multiple containers per host on the performance of HPCC's EP-DGEMM benchmark. Its performance improved (7-16%) when running one container per process combined with cgroups to restrict containers to specific cores, similar to explicitly pinning these processes. Moreover, setting CPU and memory affinity improved its performance for all tested configurations, and also made it independent on container count.

Based on the results above, it is clear that containers are capable of executing CPU-based applications with similar or equal computation performance to the native environment. Consequently, their performance impact on computation-intensive applications should be negligible. Moreover, no special configuration is necessary to achieve optimal results. Therefore, from a CPU computation standpoint, containers are a good match for HPC.

8.3 Permanent storage overhead

Bahls [6] used the *IOR* benchmark to evaluate the storage performance of Shifter containers on Cray systems. On 1 to 200 nodes, its reading and writing performances were comparable to native.

In order to determine in which conditions OS-level virtualization is more suitable for IO-bound HPC applications than hypervisor-based virtualization, Beserra et al. [11] evaluated the IO performance of LXC containers against VMs, in comparison with native performance. They utilized the `fs_test` benchmark to evaluate the MPI-IO performance of the file system on two quad-core nodes (8 GB RAM, and 1 Gb/s Ethernet). First, they executed individual instances (processes) of the benchmark on each CPU core. With a small file size (1 kB), KVM presented more stable performance for writing operations than LXC. For reading operations, LXC performed close to native execution, but its performance deteriorated with the increase in the number of containers (2 or 3 per host). KVM, on the other hand, maintained very stable reading performance, but was still slower than LXC. With a larger file size (128 MB), LXC achieved writing speeds close to native, and performance did not deteriorate when the containers per host ratio was increased. In this case, it was actually KVM that presented such performance degradation. LXC presented some oscillation in writing speed with the larger files, but its behavior was similar to native. In a second battery of tests, the authors executed a single cooperative parallel instance of `fs_test`, using all available cores. With the 1 kB file size, the performance with one container or VM per host was similar to native. With multiple virtual environments per node, however, both reading and writing performances significantly decreased, for both container and VM. For the 128 MB file size, writing performance was more stable, and with 2 instances per node LXC performed close to the native environment. For reading operations, the authors observed the same performance degradation as with the 1 kB file size. Overall, both technologies achieved near-native IO performance, which degraded as the number of virtual environments in the same host increased. Moreover, this issue was more noticeable with KVM, which presented worse IO performance than LXC in all test cases.

One of the resource allocation improvements proposed by Herbein et al. [30] deals with disk IO contention and load imbalance. They propose a two-tiered mechanism, at node and cluster level, which extends Docker and Docker Swarm to monitor the IO activity of containers. This enables considering IO load-balance when mapping containers to data center nodes, and avoiding allocating more containers on nodes whose IO bandwidth is saturated. They also implemented a mechanism to enable the Docker daemon to set IO bandwidth limits for containers. The expected

result is improved IO load balance and the avoidance of IO hotspots. This solution was evaluated by running an IO-intensive application on 90 containers mapped to 14 VMs (each with a dedicated CPU core and hard-drive). Their results show their solution was able to provide well-balanced IO bandwidth between the VMs. Without their mechanism, the same test cases presented large differences in IO bandwidth across containers.

To evaluate the storage performance of containers on an OpenPOWER system (IBM Turismo SCM POWER8 @ 3.857GHz, 8 cores \times 8 threads, 128 GB RAM, and 10 Gb/s Ethernet.), Kuity and Peddoju [43] ran the IOzone benchmark (20 times). They observed very little overhead related to IO operations due to containerization. On the other hand, executions of IOzone inside a VM displayed significant performance penalties, especially for larger files, which were attributed to inefficiencies in the VM's IO driver.

Arango et al. [3] employed the IOzone benchmark to measure the storage overhead of LXC, Docker, and Singularity. All three presented some overhead for write operations, with LXC and Singularity achieving similar performance, while Docker was the worst (37.28% overhead). For read operations, Singularity achieved almost native performance, LXC had an overhead of 16.39%, and Docker was the worst, with an overhead of 65.25%. The IO performance of random reads and random writes was also better with LXC and Singularity than with Docker. Moreover, Singularity was better than bare-metal both when operating on a contained file system and when using a bind-mounted shared NFS (*Network File System*). For random reads and writes to a shared NFS, LXC and Docker were worse than bare-metal, with LXC being slightly worse than Docker. For random reads and writes from a contained file system, LXC was faster than bare-metal, while Docker presented significant overheads. The authors say these bad results happened because they used Docker's default multilayered file system (AUFS).

Chen et al. [19] evaluated the storage overhead of their *Build and Execution Environment (BEE)*. For this, they measured the reading and writing performance of Linux's `dd` command with file sizes from 1 MB to 1 GB. The overhead of *BEE-Charliecloud* was only 0.08% for both read and write operations. *BEE-VM* was slower, with averages overheads of 15.2% for reads and 17.1% for writes. For *BEE-AWS* the averages were 0.4% for reads and 0.3% for writes. Finally, the overheads for *BEE-Openstack* were, on average, 5% for reads (1.7-7.6%), and 4.1% for writes (1.7-6.4%).

Belkin et al. [7] evaluated the IO performance of Shifter jobs on the *Blue Waters* supercomputer. They ran the IOR MPI-IO benchmark on 16 nodes (7 cores each). Their results suggest that there is no substantial difference in reading and writing performance between the Shifter and the native *Cray Linux Environment (CLE)*.

The performance overhead of Docker graph storage (used to store and manage images, containers, and other metadata), was explored by Sparks [81]. More specifically, they dealt with the case of diskless HPC hosts, which use a network-accessible parallel file system for storage. They discussed three solutions, *OverlayFS mounted over tmpfs* (in memory), a new OverlayFS implementation using a loopback XFS to mount files from a shared parallel storage, and a hybrid loopback XFS and tmpfs approach where directories with high performance

requirements were mounted on tmpfs. The *hybrid approach* presented performance close to the native OverlayFS on top of a local Btrfs. The *loopback XFS* device presented slight better than native performance for image download, but its writing performance decreased.

Abraham et al. [1] studied the impact of HPC container solutions in the IO throughput of HPC *parallel file systems* (PFS). They executed a series of benchmarks on top of the *Lustre* PFS. Container images with the configured benchmarks were executed using Docker, Podman, Singularity, and Charliecloud. They detected some startup overhead for Docker and Podman, and network overhead at startup time for Singularity and Charliecloud. Despite this, they concluded that “*the observed throughput of containers on Lustre is at par with containers running from local storage.*” Other insights included that Singularity had the best start-up times, and that Singularity and Charliecloud have equivalent IO throughput. Charliecloud, however, incurred on more PFS operations (metadata and IO).

To evaluate the storage overheads of Singularity and Charliecloud, Ruhela et al. [74] employed the *NAS BTIO* (BTIO.mpi_io_full) and IOR benchmarks. The BTIO tests evaluated Charliecloud with three different problem sizes (classes C, D and E) and up to 14K processes on two different clusters. Results showed negligible or non-existent overhead in most cases, except for an outlier at 224 processes, whose IO-time was $\approx 8.95\%$ longer than native, even if the total time was only $\approx 1.72\%$ longer. In the IOR experiments, Charliecloud was evaluated with up to 114K processes, and Singularity with up to 4098 processes. Both presented similar to native read and write throughputs.

Huang et al. [36] analyzed the feasibility of employing IO workload management tools to handle issues caused by IO-intensive workloads running inside containers. They claim that while the existing tools can address issues for conventional HPC applications, “*none of them were designed and tested to accommodate the cases of such isolated environments.*” For their feasibility study, they used the *Optimal Overload IO Protection System* (OOOPS), and the chosen container runtime was Singularity. Making the OOOPS library work with the containers was relatively easy when the OS distribution inside the container was the same as on the host. For three other containers with two different versions of a different distribution, however, they had to employ more advanced strategies. Moreover, a different strategy was required for each container. Their evaluation process employed OOOPS’s `test_stat` command for small IO tests, and an *ad-hoc* MPI-IO test designed to mimic the behavior of a real containerized HPC application. For the latter, they ran 128 MPI tasks on 4 compute nodes of the *Frontera* supercomputer, and employed a Lustre parallel file system. Results of the small IO tests using 9 different container images, demonstrated that they successfully enabled OOOPS for a range of container environments. The MPI-IO tests with unlimited IO for two different containers presented comparable execution times to the native environment (≈ 120 s). Moreover, tests with two different levels of IO throttling also presented similar containerized and native execution times. According to the authors, this indicates that OOOPS worked properly for containerized MPI-IO running across multiple nodes. Besides this evaluation, this work presented a practical exampled where they used a containerized version of LAMMPS to perform IO-intensive molecular dynamics simulations on *Frontera*.

The results of this test demonstrated that the throttling mechanism in OOOPS works effectively in both native and containerized environments.

Overall, the results we summarized above show that container technology can achieve storage (IO) performance comparable to native. Moreover, they represent a remarkable improvement compared to hypervisor-based virtual machines. To achieve the best performance, however, some aspects need attention. One of them is to avoid IO intensive operations over Docker's overlay file system (AUFS). It is probably better to mount a host directory inside the container, either as a volume or via a bind mount. Another aspect to consider is the number of containers mapped to each physical host. As discovered by multiple works, IO performance tends to degrade as this number increases, especially with small files.

8.4 GPU computation overhead

Arango et al. [3] chose *NAMD* as a benchmark to evaluate the GPU computation overhead of LXC, Docker, and Singularity. The experiments took place in an 8-core machine equipped with an NVIDIA *Tesla K200m* GPU. LXC got very close to native performance, while Docker (nvidia-docker) and Singularity achieved better performance than native execution (by a small margin).

Khan et al. [41] presented the experiences of the *EU H2020 CloudLightning* project on using containers to provide HPC applications as a service. Their focus was on applications that exploit computation accelerators of various types. Regarding GPUs, they measured the performance of the widely used *BLAS* linear algebra library. They experimented with two different builds of *GEMM*, both on bare-metal and inside a Docker container (using the *nvidia-docker* wrapper). One version of *GEMM* was linked with OpenBLAS, a CPU-based BLAS implementation. The second used cuBLAS, which employs *CUDA* to execute in NVIDIA GPUs. The testbed was a 48-core computer node, with 64 GB of RAM and one NVIDIA *Tesla P100* GPU. The bare-metal executions performed significantly better than the containerized ones, while the cuBLAS version presented the largest containerization overheads (1.1–1.27×).

Gomes et al. [26] evaluated the containerized performance of GPU-accelerated applications using a CentOS 7.3 and an Ubuntu 16.04 container image. The first application was DisVis, a software package for modeling biomolecular complexes. Executions on Docker and udocker were faster than native, with gains of 1–2% for CentOS, and $\approx 5\%$ for Ubuntu. For the second application, GROMACS, the CentOS containers were 3–5% slower than native execution for both Docker and *udocker in F3 mode* (based on LD_PRELOAD, employing fakechroot and modifications to ELF headers). With Ubuntu, the containerized performance was the same as native. Tests with *udocker in P1 mode* resulted in up to 22% performance penalty. The authors say the large performance loss in this mode is due to the elevated number of system calls caused by context switches (hybrid OpenMP and GPU execution). Moreover, they explain that newer versions of udocker mitigate this issue by filtering in only the relevant system calls.

Hu et al. [35] evaluated the GPU overhead of Singularity for three real applications. Their experiments took place in a single 56-core node, with 128 GB of RAM, and two NVIDIA *Tesla P100* GPUs. The first application was *NAMD* (*NANoscale Molecular Dynamics*), with timestep sizes 100 ps and 10 ps, executing on both GPUs, with 4, 8, 16, 32, and 48 CPU threads. With the 100 ps timestep, the overhead was from -0.812% (4 cores) to 0.807% (24 cores). For the 10 ps timestep, the overhead was from -1.675% (32 cores) to 0.843% (48 cores). The second application was the GPU version of *VASP*, whose containerized executions achieved negligible performance differences from native execution ($< 0.05\%$ with one GPU, 0.313% with two GPUs). The third GPU application was *AMBER*, a package of programs for molecular dynamics simulations of proteins and acids. For the tested input data, the execution on one GPU lasted 57,072 s in Singularity and 57,775 s natively. With two GPUs, Singularity presented an execution time overhead of $\approx 4.135\%$. For the *NAMD* and *VASP* experiments, the authors concluded that the performance of GPU applications in Singularity containers is very close, or basically similar, to native application. For *AMBER*, they concluded that the overhead is acceptable considering the total execution time.

Grupp et al. [27] evaluated the usage of optimized containers built from official TensorFlow (TF) images, to test and deploy neural-network applications. They trained two neural network architectures, Resnet-50 and AlexNet, with synthetic data and with the ImageNet data set. Their experiments employed two multi-GPU systems, one equipped with four NVIDIA *Tesla K80* GPUs (12 GB VRAM), and another with four NVIDIA *GTX 980 Ti* (6 GB VRAM). First, the benchmarks were executed inside a container instantiated with udocker [26] (briefly described in Sect. 13), using 1, 2, and 4 K80 GPUs. The results using synthetic data closely matched the official TF benchmark results for the same GPU. Compared to the executions with synthetic data, no performance penalty was observed for Resnet-50 using the ImageNet data set. For AlexNet, however, there was clear performance degradation with two and four GPUs. The authors explain that this degradation also shows up in the official TF results, but at a smaller scale. Nonetheless, they attributed this difference to the low throughput of their network storage, instead of GPU overhead. Following these experiments, they compared the udocker and Singularity container performances of Resnet-50 and AlexNet using the ImageNet data set. Their results did not indicate any significant performance difference between the two runtimes. Next, they evaluated the two neural networks with ImageNet data on 1, 2, and 4 NVIDIA *GTX 680 Ti* GPUs, achieving slightly more than double the performance obtained in the K80-based setup. In another experiment, to detect differences in accuracy between different GPU counts, they performed long-term training tests of Resnet-50 (ImageNet data set). Executions with udocker, on both K80 (5 epochs) and GTX 980 Ti (10 epochs) setups, did not show any differences between executions with 1, 2, and 4 GPUs. Moreover, the training time in the K80 setup showed very good linear scaling, while the 980 Ti setup had very similar scaling to the shorter runs. Finally, in an evaluation of the training loss in the 980 Ti setup, they found very similar behaviors in the learning curves for all three GPU counts.

Muscianisi et al. [62] evaluated the performance overhead of containerized TF using GPU accelerators. For this purpose, they built a Singularity image from the official TensorFlow Docker image and used it to train different neural networks, using the ImageNet data set. To compare the containerized and the officially reported TF performance, they ran Resnet-50 with batch size 32, using 1 to 3 NVIDIA K80 GPUs. Then, to compare containerized and bare-metal executions, they trained six different neural networks, using 1, 2, 3, or 4 GPUs, with batch sizes 32, 64, and 128 per GPU. In both experiments, the Singularity container did not introduce any relevant performance overhead.

Newlin et al. [63] analyzed the performance trade-offs of executing AI workloads in a containerized environment. They compared the performance of benchmarks from MLPerf running natively and in a Singularity container. The experiments employed two single-GPU nodes, equipped with NVIDIA *Tesla P100* GPUs, and three multi-GPU nodes, one with $8 \times$ NVIDIA *Titan V*, another with $4 \times$ NVIDIA *Titan X*, and a third one with $4 \times$ NVIDIA *GTX 1080 Ti*. The evaluation metric was *time-to-accuracy* (TTA), which is the time the benchmark takes to reach a target level of accuracy. The *Sentiment Analysis* and *Translation* benchmarks showed no statistically significant difference between container and native execution. On the other hand, *RNN Translator* showed statistically significant overhead for Singularity on multi-GPU executions. Moreover, *Recommendation* showed slightly statistically significant overhead for native single-GPU execution (i.e., native performance was worse than in the container). This last benchmark was not evaluated over multiple GPUs.

Ruhela et al. [74] evaluated GPU-based executions of MILC and VPIC on Singularity and bare-metal. Both of them were evaluated on a cluster with have four NVIDIA *V100* GPUs per node, while VPIC was also evaluate on a cluster with four NVIDIA *Quadro RTX 5000* GPUs per node. Both clusters are equipped with InfiniBand interconnects, and all executions utilized 256 GPU. MILC on the V100 cluster presented less than 4% overhead for singularity, while VPIC presented a slight overhead on the V100 cluster ($\approx 1.96\%$) and a negligible overhead on the RTX 5000 cluster ($\approx 0.65\%$).

Lee et al. [49] evaluated the GPU-based performance of three deep-learning workloads and three HPC workloads when using *gShare*, their proposed framework for managing GPU memory allocation when multiple containers share the same GPU. Their results displayed negligible performance overhead when using *gShare* compared to *nvidia-docker* without *gShare*.

As seen above, container technology is capable of providing similar GPU computation performance to the native environment. Moreover, among the few exceptions, at least one was due to issues unrelated to containers (network file system). Therefore, we can conclude that containers are a suitable environment to execute applications that rely on GPU accelerators.

8.5 Memory overhead

As part of their evaluation of containers in an OpenPOWER platform, Kuity and Peddoju [43] ran 20 executions of the STREAM (array size 23283 GB) and *RandomAccess* benchmarks (from HPCC) on a system equipped with an IBM Turismo SCM POWER8 CPU (3.857GHz, 8 cores \times 8 threads, 64 kB L1 data cache, 32 kB L1 instruction cache, 512 kB L2, 8 MB L3), 128 GB of RAM, and 10 Gb/s Ethernet. Among the three tested variants, EP-STREAM and *SingleSTREAM* (single thread) obtained similar performances in both container and bare-metal environments, with VM slight behind these two. *StarSTREAM*, however, presented performance degradation as the thread count was increased, due to contention in the shared L3 cache. In addition, both the regular *RandomAccess* and the *StarRandomAccess* benchmarks scaled well in all three environments. For small thread counts, *MPIRandomAccess* achieved slightly higher performance on bare-metal. As the number of threads increased, however, its bare-metal and container performances converged. The VM environment, on the other hand, presented performance overheads up to 52.6% (32 threads) when running *MPIRandomAccess*.

Arango et al. [3], analyzed the memory performance of LXC, Docker, and Singularity. The latter achieved slightly higher memory bandwidth than bare-metal on STREAM, while LXC had a significant overhead and Docker was the worst (\approx 36% average overhead).

Brayford et al. [12] detected negligible memory overhead when using Charliecloud to run neural networks such as Resnet-50 and AlexNet.

Sande Veiga et al. [77] evaluated the memory bandwidth of Singularity containers through experiments executed on 48 cores of the FinisTerra II cluster. The results of STREAM, with a non-cacheable array size, showed negligible bandwidth differences between containerized and native executions.

Besides pure CPU performance, Torrez et al. [84] evaluated the impact of Charliecloud, Shifter, and Singularity on memory performance. In addition to running STREAM, they separately tracked the memory consumption by the container runtimes and by HPCG. They presented similar performance results in all four environments, but memory consumption between container runtimes differed by hundreds of megabytes. Nevertheless, the authors considered the differences irrelevant, representing at most 1-2% of the total available RAM.

Hu et al. [35] also employed STREAM, in their evaluation of Singularity. Their results show very close containerized and native performances, with a maximum overhead of 0.506%, for the copy operation.

Besides their CPU overhead evaluation, Gantikow et al. [24] employed the STREAM benchmark to analyze the memory performance of Podman containers. Their results show less than 1% memory bandwidth difference between the native, Singularity, and Podman environments. Thus, indicating no significant memory overhead from containerization.

As part of their performance evaluation of medium and large scale containerized workloads, Ruhela et al. [74] evaluated the memory consumption when

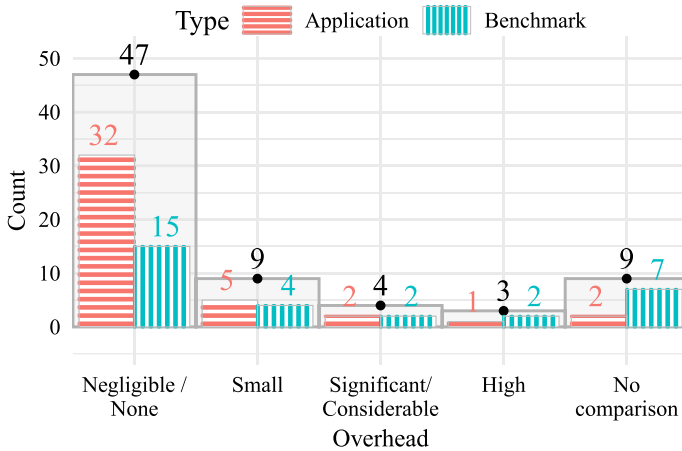


Fig. 4 Application and benchmark experiment counts by level of overhead. The shaded area represents the total counts

running MILC with up to 14K processes on a cluster composed of 56-core nodes. Their results for Charliecloud containers were nearly identical to native.

Liu and Guitart [52] measured the effects on memory bandwidth from running multiple containers per nodes, with Docker and with Singularity instances. They ran the HPCC’s EP-STREAM benchmark on a five node HPC cluster with an InfiniBand interconnection, while increasing the number of containers until there was one process per container. Their results showed that EP-STREAM performed similarly in all test cases.

The experiments summarized above show that containers can provide near-native memory (RAM) performance to the encapsulated applications. Even though there is not a clear reason for containers to cause this type of overhead, there were a few experiments that measured significant overheads. One of these results [43] was due to contention in the L3 cache when increasing the number of threads of *STREAM*. It is not clear, however, what would cause this issue to be more exacerbated when inside the container. Another work [3] encountered significant overheads for LXC and Docker containers, but not for Singularity. In this case, Singularity’s advantage maybe due to its more permissive default configuration. If so, it may be possible to improve the performance of LXC and Docker by configuring them to behave more similar to Singularity.

9 Overall application overhead

In the previous section, we discussed experiments whose goal was to evaluate the impact of containers on the performance of operations related to specific system resources. Now, we present experiments whose goal, as stated in the respective work, was to evaluate the performance overhead on the application as a whole. In this context, we use the term “application” to refer either to a synthetic benchmark or

Table 2 Application-based experiment counts by level of overhead

	High	Significant/ considerable	Small	Negligible/none	No comparison	Total
WRF		1 [82]		2 [35, 89]		3
Quantum Espresso				3 [6, 77, 81]		3
GROMACS			1 [9]	3 [44, 67, 89]		4
MILC		1 [74]		2 [73, 89]		3
NAMD				2 [35, 89]		2
VPIC					1 [19]	1
VASP				1 [35]		1
SNAP				1 [8]		1
POP2				1 [6]		1
PISM				1 [6]		1
OPM-based upscaling					1 [41]	1
openQCD			1 [26]			1
OpenFOAM DepthCharge2D			1 [61]			1
OpenFOAM				1 [67]		1
NEURON				1 [67]		1
MiniMD				1 [8]		1
MiniAMR				1 [8]		1
MiniFE				2 [8, 75]		2
MEGADOCK				1 [2]		1
MasterCode			1 [26]			1
LS-DYNA				1 [24]		1
LAMMPS				1 [47]		1
"a genomics application"				1 [41]		1
COSMO			1 [9]			1
Bowtie		1 [4]				1
Alya FSI code				1 [72]		1
Alya				1 [72]		1
3DGAN				1 [12]		1
bwa-mem2				1 [91]		1
samtools				1 [91]		1
GATK4				1 [91]		1
Grand total	1	2	5	32	2	42

to a real, full-fledged, application. It is important to note that some applications may also appear in other sections, since they may have been used in experiments whose goal was resource-specific (e.g., to compare different networking configurations).

As we have seen in Fig. 3, 35 of our selected works tackle this challenge. Thus, it represents the largest portion of the overhead-related works. Moreover, each work may include many of such experiments, which leads to a total of 72 experiments (42 applications and 30 benchmarks). Looking at the complete set, however,

Table 3 Benchmark-based experiment counts by level of overhead

	High	Significant/ consider- able	Small	Negligible/none	No comparison	Total
NPB		1 [97]		2 [19, 80]	1 [53]	4
HPL	1 [21]			1 [8]	2 [20, 45]	4
HPCG			1 [8]	3 [75, 84, 93]		4
Graph'500	1 [21]			1 [97]	1 [20]	3
KMI Hash				2 [8, 75]		2
UTS					1 [53]	1
Terasort		1 [61]				1
Teragen			1 [61]			1
t_cnn_benchmark (TF)			1 [9]			1
Stencil2D (Charm++)					1 [53]	1
Poisson LU (FEniCS env.)				1 [28]		1
Poisson AMG (FEniCS env.)				1 [28]		1
MPI Poisson AMG (FEniCS env.)				1 [28]		1
ad hoc matrix multiplication				1 [71]		1
Jacobi1D (Charm++)					1 [53]	1
HPGMG-FE				1 [28]		1
HPCC				1 [51]		1
ad hoc emb. parallel code			1 [71]			1
Grand total	2	2	4	15	7	30

we began to see similarities among most results. Therefore, to avoid having a disproportionately long section, we decided to summarize these works based on the benchmarks or applications they employed, and on the measured overheads.

The total number of application and benchmark containerization overhead experiments is displayed in Fig. 4. Additionally, in Tables 2 and 3, we list the experiments that employ each application or benchmark. They are identified by a citation to the work where they were performed, and classified into four levels of overhead, corresponding to the best result found in that work. Considering an overhead o the levels are (1) *High* ($o > 15\%$), *Significant/Considerable* ($5\% < o \leq 15\%$), *Small* ($2\% < o \leq 5\%$), and *Negligible/None* ($o \leq 2\%$). As a remark, we note that 9 experiments, from 4 different works, did not compare the performances of container-based and bare-metal executions.

Looking at this data, it becomes clear is that in the majority of experiments, $\approx 78\%$ (56 of 72), container-based executions presented negligible or small performance overheads over bare-metal. Moreover, if we disregard the 9 experiments that did not actually evaluate the overhead over bare-metal, this proportion grows up to $\approx 89\%$ (56 of 63), with $\approx 74.60\%$ measuring negligible or absent overheads, and $\approx 14.29\%$ measuring small overheads.

Only 7 of the 72 experiments ($\approx 9.72\%$) resulted in significant or high overheads, representing only $\approx 11.11\%$ of the ones that actually compared containers with

bare-metal. Below, we discuss these experiments and try to explain these overheads, when possible.

Azab [4] evaluated the containerization overhead of *Bowtie*, a short DNA sequence aligner, running 2, 4, 8, 16, and 32 processes on two 16-core nodes. Its container-based executions incurred high overheads when executing more than 2 processes, with both Docker ($\approx 26.9\text{--}70\%$) and Socker ($\approx 42.3\text{--}70\%$). Interestingly, Socker performed similar to Docker when executing 1, 2, 16, and 32 processes. At first glance, these overheads may seem really high, but the presented execution times include the container deployment, start-up and removal times. Therefore, these results do not represent the actual performance of the application. Moreover, the container images were removed at the end of each job and, consequently, redeployed for every new container job.

In another experiment, Steffene et al. [82] measured significant overheads (up to 7.8% with 6 processes) running *WRF* (*Weather Research and Forecasting* model) inside containers, even though other works [35, 89] managed to achieve negligible overheads for this application. By analyzing execution traces, Steffene et al. [82] noticed that a long time was spent on `MPI_Wait` events. Therefore, these overheads are likely related to communication. They attributed this issue to the *Docker Overlay Network* they employed in their Docker Swarm container cluster.

Among the experiments with benchmarks, Chung et al. [21] found high overheads when running *HPL* inside Docker containers. This is unexpected, since containerization should not significantly slow down to such a computation-intensive benchmark. They compared bare-metal, Docker, and VM-based executions with different input sizes, on two nodes equipped with an Intel Xeon 2670 CPU (2.6GHz, 8 cores, 16 threads), 64 GB RAM, and 1 Gb/s Ethernet. Since all configurations were tested in the same nodes, and *HPL* is not communication intensive, the slow interconnection should not be a major source of overhead. The real issue seems to be that *HPL* was executed with 32 processes, which is double the number of CPU cores in their testbed. Even though the two CPUs can simultaneously run 32 threads, via *simultaneous multi-threading* (SMT), such an optimized compute-intensive application should keep the CPU's functional units busy most of the time. Therefore, leaving little opportunity for multi-threading. This work also evaluated Docker-based executions of *HPL*, with 32 processes, and increasing container counts. This resulted in increasing performances up to 16 instances (one per physical core). With 32 instances, however, performance went down by $\approx 61.5\%$. Another interesting result from this work is that Docker was able to run *HPL* with input sizes larger than 85% of the available RAM, which was not possible inside a VM.

In another experiment, Chung et al. [21] measured high overheads for the execution of *Graph500* inside a container. They evaluated its execution with graph scales from 20 to 26 (controlled by the variable `SCALE`). Overall, both Docker and bare-metal performances slowly increased for scales 20–22, remained mostly stable for scales 20–22, and significantly increased for scales 25 and 26. The Docker execution overhead, however, significantly changed between different scales. The largest variability was found in the 20–24 interval, where it goes up and down inside the $\approx 3.8\text{--}13.3\%$ range. For larger scales, the performances for both Docker and bare-metal significantly improve, but bare-metal presents higher, seemingly linear,

speedup up. Docker's performance, on the other hand, does not increase as much, and further decelerates when going from SCALE 25 to 26, with respective $\approx 15.9\%$ and $\approx 34.24\%$ overheads. The process count for these experiments is not specified by its authors. Therefore, we assume it is the same as in their previous experiments (HPL). So, they may have, once again, relied on SMT to run double the processes as the available CPU core count, which could be causing the high overheads. Nevertheless, different from HPL, increasing the number of Docker instances had no negative effect in the performance of Graph'500. In fact, the best performance was achieved with 32 instances (one process per container). A final interesting result from this experiment is that Docker performed much closer to the VM than to bare-metal, only distancing itself for scales 25 and 26.

Besides experiments with Graph'500, Zhang et al. [97] observed significant overheads in Singularity-based executions of NPB (class D) (512 processes) on their Haswell platform. Among the programs included in NPB, the overheads were $\approx 5.66\%$ for CG, $\approx 5.88\%$ for MG, and $\approx 6.14\%$ for LU. BT and SP were not executed, and the plot in the paper does not display any visible performance difference between bare-metal and container executions of EP, FT, and IS (the actual values were not included in the text). Unfortunately, no explanation was provided for the higher overheads for CG, MG, and LU.

Muhtaroglu et al. [61] observed significant overheads when running the *Terasort* benchmark (random number sorting using *Hadoop*), on *Scaleway* C2S (4 cores) and C2M (8 cores). Docker presented a significant overhead of $\approx 6.9\%$ (65 s) for the 10 GB input size. In contrast, with a 20 GB input, it presented a small overhead ($\approx 2.6\%$). Moreover, on the same platform, they achieved small overheads for both *Teragen* ($<5\%$) and *OpenFOAM DepthCharge2D* (3–4%). In another experiment, they ran two concurrent 10 GB *Terasort* jobs led to $\approx 6.6\%$ shorter average execution times than the 20 GB bare-metal case. On the other hand, the same strategy led the IO-intensive *Teragen* to run 32% slower than the 20 GB configuration. Thus, indicating the need to consider IO-intensiveness when mapping containers to hosts. Unfortunately, the authors do not explain the significant overhead for the 10 GB *Terasort*. Nevertheless, based on the fact that the 20 GB case obtained smaller overheads, we formulate the hypothesis that the larger the problem size led to better overlapping of computation and IO operations. If true, this would indicate Docker is causing some kind of IO-related overhead (e.g., communication, storage). It is also unfortunate that the sample sizes for these experiments is not provided. Therefore, we cannot adequately infer the accuracy of the provided averages.

Ruhela et al. [74] found significant overheads for Charliecloud on tests with CPU-based MILC running up to 140K processes at a cluster (28 cores \times 2 sockets per node @ 2.7GHz, 100 Gb/s InfiniBand). This overhead was less than 10% at various system scales, but this includes setup and teardown costs. According to the authors, these would be amortized in real-world executions, "where MILC is allowed to run for multiple trajectories and several steps per trajectory."

Based on the results in this section, the overall conclusion is that containers are capable to execute HPC applications with negligible or small performance overheads compared to bare-metal execution. This is demonstrated by the fact that only

9 of the 63 experiments (from 32 works) that evaluated this aspect resulted in significant or high overheads.

In most cases where results indicated higher overheads, their causes were not directly related to containerization. Among these, in one work [4] the experiments were designed to aggregate the execution time of the application with the container deployment and startup times. Moreover, previously deployed container images were purposefully made non-reusable. Similarly, Ruhela et al. [74] included the container setup and teardown times in their measurements. In another case [82], the technology or configuration choices were suboptimal for the test scenario. In addition, at least one work [21] relied on hardware characteristics that may have caused performance degradation. Finally, one work [61] presented an incomplete statistical analysis, and another [97] did not provide enough information to explain the significant overheads they encountered.

10 Other overhead

Bahls [6] evaluated the startup times of containerized versions of PISM, Quantum Espresso, and POP2 on Cray systems. They tested two techniques to access the host's Cray MPI. In general, the versions that rely on MPICH-ABI compatibility presented longer startup times than the one that included Cray MPI libraries inside the container. The authors mention, however, that the lower performance of the MPICH-ABI approach may be due to a workaround that requires copying some libraries to a Lustre file system.

Higgins et al. [31] evaluated the deployment overheads of their *virtual container cluster* (VCC) model, which is based on Docker and uses *Weave* to create an overlay network. Their results show that VCC introduces overhead at deployment, with the longest portion of time being spent on pulling the container image. Moreover, their virtual node discovery process also added some overhead.

Sparks [80] evaluated the performance of Shifter, Singularity, runC and *rkt* containers on Cray systems. Among other aspects, they analyzed the startup time compared to the native *Cray Linux Environment* (CLE). Shifter and Singularity performed well as the number of nodes increased, while runC had the largest startup cost, because a complete image has to be copied to the host prior to execution. *rkt* also took longer than the first two, due to having to create a per-node instance of the container, in tmpfs. The authors remark on the possibility to improve the startup time of runC by pre-fetching image data.

Arango et al. [3] measured the time to execute a simple operation (*echo "Hello world!"*), on LXC, Docker. This includes starting, executing, and stopping the container. According to the authors, the shutdown operation of the container is the slowest. Based on their graphics, it seems that Docker is the slowest to run the full set of operations (using the `docker-run` command), while LXC is slightly faster. It also seems that the time to run the command on an already running container is negligible for both runtimes.

Belkin et al. [7] evaluated the startup time of Shifter container jobs launched via a specially configured PBS *resource manager*. This can be either automated by

PBS, or done by passing the arguments directly to Shifter. They tested a *user-defined image* (UDI) with 26 MB and another with 1.7 GB. First, to evaluate how job startup time depends on the number of nodes, they launched a Shifter job that exploits only one processor per node. The results suggest that the startup time of a Shifter job is practically independent of UDI size. For less than 256 nodes, they found a sublinear dependence of the job startup time on the node count, which became linear beyond 256 nodes, and superlinear beyond 2048 nodes. Their second experiment used 80 nodes to evaluate the dependence of Shifter job startup time on the number of MPI ranks per node. Their results, once again, displayed startup times practically independent of UDI size. When UDI arguments were specified to PBS at job submission time, the startup times were the same, regardless of the number of processes per node. When passing the arguments directly to Shifter, however, the startup time became dependent on the number of MPI ranks per node.

Ramon-Cortes et al. [71] evaluated the deployment of *MatMul* (a matrix multiplication code), and of an embarrassingly parallel application in a KVM cluster managed by Openstack, a Docker cluster built with Docker Swarm, a third one built with *Mesos*, and on an execution with Singularity. KVM's creation phase takes 109 s, and its deployment takes ≈ 30 s in the best case, or ≈ 98 s if the image needs to be downloaded. For Docker with Openstack the creation phase took 24 s with the base image available, and up to 82 s otherwise. The deployment phase for the Docker setup took from ≈ 5 s, when all layers were available in the Docker engines, and up to ≈ 98 s when they were not. The Mesos setup used its default Docker execution, thus the creation and deployment phases lasted the same as in the previous result. The creation phase for Singularity is the same as for Docker, but, the image must be imported and converted prior to execution. With the image ready, the deployment is considerably faster than Docker's.

The compilation overhead of MasterCode in a containerized environment was evaluated by Gomes et al. [26]. This complex application interfaces several scientific codes, and is directly dependent on a wide range of tools and libraries. Moreover, it includes several other applications and their dependencies. The average compilation times for *udocker* in P1 mode (based on *ptrace*) were $\approx 30\%$ longer than on the native environment. By using a newer version of *udocker*, which implements *SECCOMP filtering* to reduce the number of traced system calls, this performance loss was reduced to $\approx 5\%$.

Muhtaroglu et al. [61] evaluated the deployment overhead of Docker containers employed to run a CPU-intensive workload (OpenFOAM CFD). They observed deployment times 10–15 \times shorter than with VMs. Thus, concluding that “*dockerized*” HPC can be set-up and deployed much faster than VM-based alternatives. They also remark that any remaining potential performance gains in containerized application execution would be negligible compared to this huge improvement in deployment times. Besides, they add, this enables a quick setup-test-reconfigure-retest cycle, thus removing IT bottlenecks and, consequently, allowing engineers and scientists to solve real scientific and sectoral problems.

Rudyy et al. [72] evaluated the deployment overhead of Shifter, Singularity, and Docker. The first two presented negligible overheads, but Docker's was much higher (103 s, for 112 MPI processes).

In an evaluation of the execution of real-world scientific applications on Singularity, Wang et al. [89] also took time measurements outside the container (Singularity startup, image loading, application startup, IO, and so on). They detected a big overhead for loading the largest image, which includes a built-in copy of the Intel compiler. For other two images, which mount-bind the host's compiler and MPI, this overhead was negligible. Additionally, two of the applications presented much slower start-ups on the image without binds than on the other two. Therefore, the authors recommend keeping container images small. Finally, they observed that increasing the node count did not affect the overall overheads.

As part of their evaluation of containerized HPC on an ARM-based cluster, Peiro Conde [67] compared the energy consumption of container-based and bare-metal executions of GROMACS. They concluded that there is almost no difference in energy consumption between the two. Besides that, a very basic test with a single container executing only the `sleep` command, demonstrated an average startup time of 6 seconds for Singularity. Another experiment, creating several MPI ranks that, once again, only execute the `sleep` command, has shown increasing startup times for Singularity as the number of ranks increases. It is worth noticing, however, that this increase is sublinear (from ≈ 6 s to ≈ 17 s when going from 1 to 256 cores), and that the startup time for bare-metal also increases, albeit at a smaller rate.

Besides evaluating communication and overall execution overhead, Ruhela et al. [73] measured the total memory consumption when running the *MILC* benchmark natively, and in a Singularity or Charliecloud container. Those experiments launched 69, 984 processes on 1296 nodes of the Frontera supercomputer. The authors observed similar memory utilization for all three cases, at various problem sizes.

As part of an evaluation of containers applied to medium and large-scale HPC workloads, Ruhela et al. [74] measured the container setup and teardown times, and the containerization overheads on a supercomputer workload. By running the *IMB MPI_Init* benchmark with 3584 processes (64 nodes \times 56 processes) to 229,376 processes (4096 nodes \times 64 processes), they measure setup and teardown overheads between 6% and 14% for Charliecloud. To evaluate a supercomputer workload, they simulated a typical job arrival pattern in supercomputer centers, based on data from previous works, and employing an exponential distribution to the job durations. In this experiment, Charliecloud presented up to 4% accumulated overhead for all containerized against all bare-metal runs. About this result, Ruhela et al. [74] remark that higher problem sizes and job lengths are more favorable to containers, since this results in less jobs per unit of time. Another interesting overhead presented by Ruhela et al. [74] occurred in their *IMB MPI_Alltoall* tests where, despite both Singularity and Charliecloud achieving on par communication performance with bare-metal, and similar CPU usage and memory occupation, the former incurred 10% more overheads. They attributed this difference to a higher number of page faults, context switches and file IO operations.

Okuno et al. [65] evaluated the potential increase in total throughput when running an ensemble of multiple containerized molecular dynamics (MD) simulation in parallel, measured in simulated *ns* per day. This means running multiple simulations in parallel, each one inside a different container. For this purpose, they ran two different simulations on OpenMM and GROMACS, on a single computer node (Intel

Xeon Gold 6148, 20 cores \times 2 sockets @ 2.40GHz, 2 threads per core, 96 GB DDR4 per CPU socket). First, they evaluated single container speedup with an increasing number of threads (relative to one thread). GROMACS scaled much better than OpenMM, with one of the simulations on OpenMM only scaling up to 20 threads. In a second experiment, they evaluated the effect of increasing the number of containers (up to 20) while proportionally decreasing the number of threads per container. Both OpenMM and GROMACS achieved better ensemble throughput when increasing the number of containers. Moreover, OpenMM presented a larger increase in total throughput when increasing the number of containers, due to its worse single-container scalability. On the other hand, GROMACS presented a much more pronounced reduction in speedup in each individual simulation. Last, they analyzed the effects of simultaneous multi-threading (SMT) on ensemble throughput, by mapping containers to different logical cores in the same physical CPU core. GROMACS presented no improvement, while OpenMM presented up to 2.22 times larger ensemble throughput with two containers per physical core on a total 10 cores.

LMPX [91], a container implementation aimed at composability, presented the lowest overhead for starting and immediately destroying a container compared to Singularity, Docker, udocker, and Podman. The second place was taken by Singularity, which was more than 6 times slower. Moreover, LPMX was also shown to incur in considerable overhead for software installation and compilation. Its authors argue this is not a big issue since it should be a one time cost that would be diluted along multiple executions.

Most of the experiments summarized in this section analyzed overheads on container deployment and startup. In this aspect, overall, we can see that containers perform better than VMs. Moreover, in most cases, HPC-specific implementations performed better. Therefore, we can say HPC-specific container implementations provide the best deployment times. Among the other results, by integrating Shifter with PBS, one work [7] demonstrated application startup times that are independent of container count. Another [43], has shown inconclusive results (due to different execution environments) for Shifter containers with a copy of vendor-optimized MPI compared to relying on MPICH-ABI compatibility. A third one [26] found significant compilation overhead for a physics application in an udocker container in P1 mode, but this seems to be resolved in newer versions of udocker. One specific work [89] has shown big overheads when including a large piece of software in the container, which reinforces the importance of keeping container images small. One work [73] measured similar to native memory utilization by an application running on Singularity and Charliecloud containers. In experiments with Charliecloud, one work [74] showed a small aggregated overhead for a simulated typical job arrival pattern, and considerable overhead in an `MPI_Alltoall` benchmark. Finally, a work related to genomic analysis workloads [65] demonstrated that running multiple containerized application instances can increase ensemble throughput, but the benefit is larger for applications with bad parallel scalability.

11 Application portability

Portability is arguably one of the main advantages containers bring to HPC. They allow the application a high degree of independence from the underlying platform. In other words, they enable the creation of *User Defined Software Stacks* (UDSS) [69], including all the dependencies needed to run their application. Thus, eliminating the need to install and configure these dependencies, as well as the application itself, on every system where it needs to be executed. Besides advantages in terms of system management, this feature reduces or eliminates the need for rebuilding and reconfiguration when attempting to run the application on a new system.

Weidner et al. [90] discussed several challenges faced by traditional HPC platforms in order to provide support for application portability, and for the efficient execution of what they call "*second generation applications*". These have characteristics such as (1) typically non-monolithic designs; (2) dynamic runtime behavior and resource requirements; or (3) relying on high-level tools and frameworks to manage computation, data and communication. In addition, they may explore new computation and data handling paradigms, or operate on a federated context, spanning multiple geographically distributed HPC systems. According to the authors, the barriers to running these applications in HPC systems stem from their static resource allocation policies, and the difficulty of (1) handling high volumes of data, (2) efficiently handling intermediate data, (3) inspecting the progress of executions of coupling parts of application concurrently running on different platforms, and (4) porting the applications to new platforms. After analyzing several details of these challenges and recommending approaches to mitigate them, the authors presented cHPC (container HPC), which consists of OS-level services and APIs that run alongside and can integrate with existing job schedulers, via LXC containers. This way, it can provide isolated user-deployed application environment containers, with support for introspection, and for using of cgroups for resource throttling. Applications, either containerized or native, continue being submitted via the HPC job scheduler. Container and node metrics are collected in real-time, to provide platform and application introspection. This data can be streamed to different types of consumers (e.g., users, monitoring systems, and the application itself), and is also stored in a database. The latter is used to compare platform data with user-set thresholds, and to send alarms whenever these are violated. According to the authors, a version of cHPC has been deployed in a virtual 128-core Slurm cluster deployed on 8 dedicated Amazon EC2 instances. In addition, they claimed to be in the process of deploying it in a 324-node, 1536-core cluster. Moreover, they mention the non-invasiveness of their implementation as being a key asset, and being unable to optimize the HPC jobs that run alongside the containerized applications as being a suboptimal aspect of their system.

Hale et al. [28] reported on how the FEniCs project uses containers to distribute its software, packaged together with its entire execution environment. This way, they facilitate its installation and execution, resulting in increased portability, and making it accessible to more users. Moreover, packaging the entire environment in a container also reduces the probability of crashes due to software incompatibility.

de Bayser and Cerqueira [22] developed an approach to facilitate the integration of Docker with MPI, which they implemented via a *wrapper* for the `mpirun` command of MPICH. This wrapper identifies the number of requested hosts and uses Docker Swarm to create a container cluster, where the MPI application will be executed. Container placement and load balancing is left entirely to Docker Swarm, with no user input. Unfortunately, this work does not provide any evaluation of this mechanism.

Nguyen and Bein [64] used Docker Swarm mode to automate MPI cluster setup and deployment, thus providing “a ready-to-use solution for developing and deploying MPI programs in a cluster of Docker containers running on multiple machines.” Their goals were to provide a consistent development environment for MPI projects, and a simple workflow to deploy MPI applications in a real cluster. Their solution revolves around providing a base container image that can be used to create a consistent development environment, by running it in interactive mode and mounting the project directory inside the container. For deployment, the developer would create a new layer on top of a second base image, which is pre-configured to facilitate its deployment and execution of the MPI application in the cluster, using Docker Swarm. A few provided shell scripts are used to facilitate this process.

Younge et al. [93] worked on deploying the same Docker image on a laptop, on two clusters at Sandia National Laboratory, and on a cloud environment (AWS). The idea is to configure the container on the laptop, and then make it available for other platforms to execute it using Singularity or Docker. They used a single Docker image to execute *HPCG* and *IMB*, with Singularity in the clusters, and Docker on AWS. According to its authors, this was the first work to evaluate Singularity on a Cray XC-series testbed. This required a few modifications to the Cray Native Linux environment, as well as making Cray’s custom software stack accessible inside the container.

Khan et al. [41] presented four use cases to demonstrate containerization efforts of the *CloudLightning* project. The chosen applications are either CPU-based, or employ hardware accelerators such as *FPGA-based DFEs*, *Intel Xeon Phi*, or *NVIDIA GPUs*. Their goal is to facilitate access to these, by employing an application-as-a-service model coupled with automated resource allocation based on *QoS* parameters. This approach is expected to lower the technical barriers and to greatly reduce the required time investment to start using these applications, leading to more efficient resource usage. Performance results were presented in Sects. 8.4 and 9.

Chen et al. [19] created the *Build and Execution Environment* (BEE), which aims to build similar HPC-friendly environments across different platforms. It uses standard docker images, but hides launch details from users. Additionally, it exposes access to host-level networks and can also configure shared directories. Its structure is composed of four back ends, which run dockerized applications. On HPC clusters, we would employ *BEE-Charliecloud*, while *BEE-VM* allows for hardware architecture independence, by running Docker on top of a virtual-machine. *BEE-AWS* enables execution on AWS, by using Docker on top of a standard AWS virtual machine. Last, we have *BEE-Openstack*, which uses Docker to execute the application on top of the Openstack cloud OS. The four back ends were evaluated in terms of communication bandwidth and latency, storage read and write performance, the

execution times of a CPU-intensive and a memory-intensive OpenMP benchmarks, and of a memory-bound plasma physics code (MPI and pthreads). Based on the results, the authors concluded that BEE can achieve comparable performance to native execution.

Sampedro et al. [76] presented a containerization-based *continuous integration and continuous delivery* (CI/CD) architecture to “support the development and management of scientific codes and software environments for HPC end users.” The paper presents a use case of a CI/CD pipeline to automate the benchmarking of the MFIX-Exa application. Jenkins is used to automatically pull new commits to build the application in a Singularity container, and to test the new versions. If all tests are passed, Jenkins pushes the new container into the registry. Jenkins can also automate performance tests of the new container, run scripts to plot the results, and also check correctness of the results. It may also automatically generate historical plots, to compare different versions, or to associate performance changes to specific commits. This combination of Singularity and Jenkins enables automatic code testing, facilitating the sharing of containers through the Singularity register. Moreover, Jenkins’s automation eliminates most of the manual steps in the process.

Belkin et al. [7] presented the efforts on supporting Shifter containers on the *Blue Waters* supercomputer (a Cray system). They modified the *PBS resource manager* to execute special prologue and epilogue scripts. These perform the automatic setup and teardown of container environments on the computation nodes. Collected data on the usage of Shifter on Blue Waters show that the majority of the node hours were spent by the *LIGO project*, for the execution of *High-Throughput Computing* (HTC) codes. This project uses the *Open Science Grid* (OSG), and Shifter was employed to enable them to use an OSG-ready Docker image on Blue Waters, “eliminating the need to adapt the image for each resource provider.” Other projects using Shifter on Blue Waters include *ATLAS* and *NanoGrav*, which are also based on OSG. Besides, Shifter was also used to execute the *QWalk* and *PysCF* electron structure computational physics and chemistry codes, and *PowerGRID*, an MPI application for medical resonance image construction which can leverage GPU accelerators.

Ramon-Cortes et al. [71] proposed a methodology for integrating container engines with parallel programming models and runtimes. Their prototype integrates the COMPSs (*COMP Superscalar*), a task based programming model, with containerization. This was achieved by extending its integration with cluster resource and queue managers to support the deployment and execution of containerized applications. COMPSs is responsible for the scheduling and execution of the tasks in the computing nodes. The produced prototype is able to employ different container engines to build a Docker or *Mesos* cluster, or to employ Singularity, and to transparently deploy and execute the containers in an HPC cluster. Additionally, the COMPSs prototype is able to execute either with a static or with a dynamic pool of containers. The latter enables adaptation to resource usage and availability.

Based on the experience of PRACE (*Partnership for Advanced Computer in Europe*) in supporting Singularity containers in HPC clusters, Sande Veiga et al. [77] presented a paper on using containers to deploy and integrate complex software and their dependencies. They specifically focus on maximum portability without wasting computational capacity. This work presents different approaches and configurations

for running Singularity containers in HPC environments, each with its advantages and drawbacks, in terms of portability and performance. These include using the same MPI version in container and host, or relying on ABI compatibility to support mixed MPI versions, or using MPI binding (mount binding the host's MPI libraries). They also compared using the vendor's MPI implementation versus OpenMPI (same version on host and container). On their tests, the container with Intel MPI achieved ≈ 5 times faster execution of *Quantum Espresso* than the one with OpenMPI. This reinforces the importance of hardware specific customization (e.g., vendor drivers and libraries) and optimization of containers, even if it hurts portability. In their conclusions, the authors state that the minimal performance loss from containerization is more than justified by "the time that will be saved using this novel form of software deployment."

Hu et al. [35] evaluated the portability of Singularity images, by testing a container image for the VASP application on a different server than the one it was built for. They were able to attain stable executions on the new server, with good performance results. Moreover, they tested Singularity's ability to use Docker images, by building a Singularity container from a WRF image pulled from Docker Hub, and evaluating the performance overhead of executing this application inside this container (see Sect. 9).

Piras et al. [68] presented an approach to accommodate service-oriented and container-oriented workloads into existing HPC infrastructure in a minimally disruptive way. They proposed a method to expand Kubernetes clusters running in a cloud infrastructure onto HPC resources managed by a batch-style scheduler. This was achieved by implementing a hybrid scheduling technique that enables the setup and tear-down of Kubernetes worker nodes onto HPC nodes allocated through the *Grid Engine* (GE) batch job scheduler. The authors claim this approach is already in use to run services and a WRF workflow on an always-on cluster, that is deployed on a private Openstack cloud. This allows the Kubernetes cluster to be expanded on-demand, when the resources on the Openstack cluster are insufficient to execute a workload. According to the authors, this expansion capability was used to run the WRF workflow on hundreds of MPI-connected cores.

Grupp et al. [27] proposed packaging performance-optimized versions of commonly utilized machine-learning software into containers, to serve as building-blocks for complex neural network applications. They started from an optimized container image provided by the TensorFlow project. These containers were tested by executing two different neural network models on two different multi-GPU systems, in which they displayed good speedup, and no overhead due to containerization.

Michel and Burnett [60] presented their effort at creating a base-container to provide GPU-optimized common *Compute Vision* and *Machine Learning* libraries, which can be used as the building-block for the interactive design and test of complex analytics. Their starting point was the `cuda_tensorflow_opencv` container, which contains many often used tools for ML research. They compiled various tagged versions of the container, and made these builds available in Docker Hub. The authors demonstrate how to run the container for use as a repeatable test framework, besides using it as a base for a new container, once the repeatable tests are done. They also demonstrated the integration of the *Darknet* "You Only Look

Once” algorithm (object detection and classification) into a new container. The base image, `cuda_tensorflow_opencv`, was created from the `cuda_tensorflow_opencv` image, by installing the required `cuda` packages from NVIDIA. They used this Darknet container to perform object detection on a video file and display, frame-by-frame, the bounding box and category name. According to the authors, this demonstrates the easy abstraction of the GPU-optimized Docker image it is built from. They also remark on the building-blocks approach as being “*the right one to design a reproducible and reusable video streaming ML & CV HPC solution.*” Moreover, they note that it can be used as the foundation for novel analytics, reducing the technical barrier of entry, and allowing researchers to focus on the analytics instead of system deployment.

Youn et al. [92] used Singularity containers to distribute their *DARE-MetaQA* framework to nodes of two different HPC clusters belonging to two institutions. This framework employs meta-learning neural network models, and aims to enable large-scale question-answering. It uses *PyTorch* and *Dask* to enable the concurrent execution of multiple models on distributed resources, and to take advantage of multi-GPU environments. They chose to use containers as a means to lower development costs, by not having to repeatedly deal with dependency management and compatibility issues that arise from using multiple HPC systems.

Community Collections (CC) [56] combines Singularity with *Lmod* to enable the sharing of software stacks across different HPC sites. Its goal is to reduce the redundancy in building software required by researchers at different sites. It handles the deployment of CC, of its dependencies, and of Lua extensions to *Lmod*. The latter enables the creation of versionless modules, whose software packages can be automatically updated as soon as a new version becomes available (using `nvchecker`). CC is designed to work side-by-side with the *Spack* and *Easybuild* package managers. It relies on community maintained sources to retrieve and deploy containers in order to fulfill user requests. More specific software installation is handled by *Lmod*. In addition, pre-built containers can be shared between HPC sites. According to the authors, besides HPC systems, CC is effectively deployed on cloud resources, having been tested on *Google Cloud Platform* resources. An important limitation of CC is that it does not provide an easy way to handle hardware-specific optimizations. In this case, CC trades-off performance for portability.

Canon and Younge [14] discussed the obstacles to achieving the portability promised by containers in HPC, as well as potential solutions. They argue that the mechanisms to achieve flexibility, portability, and reproducibility, through containerization, come at the cost of performance. Moreover, that the current solutions, such as bind-mounting host libraries, have result in a trade-off between portability and performance. They also highlight the lack of interoperability between container runtimes, especially in the way they handle the mapping of host libraries into containers, and the incorporation of specialized hardware. In relation to the bind-mount approach, they point-out that differences in implementation and in user interfaces make it difficult, or even impossible, to use the same container image with different runtimes. In addition, they discuss issues due to the reliance on ABI compatibility between host and container libraries. Moreover, they highlight a series of circumstances where these two approaches may fail. As a solution, Canon and Younge [14]

proposed using the label capabilities of the OCI image format to communicate container requirements to the runtime, in a standard way. With this in place, if the runtime can meet the requirements, it would automatically perform the necessary configuration at initialization. To demonstrate this approach, they employed a Python script to retrieve metadata from an image and extract labels that were used to select the appropriate Shifter modules for the container. In addition, they discussed potential solutions to the reliance on ABI and version compatibility between injected libraries and the container images. The most promising one, according to them, would be a container compatibility layer, in the form of a lower-level runtime library that would provide access to specialized hardware. They highlight, however, some difficulties in making this solution gain traction, like the need for all HPC container runtimes to agree to leverage this tool, and for support from various vendors. They also mention a *dual virtualization* approach, with containers on top of a lightweight hypervisor-based VM. It is not clear, however, if the added performance overhead would not make this solution unsuitable for HPC.

To facilitate the execution of applications built with different versions of the *Cray Processing Environment* (CPE) on the same system, Martinasso et al. [57] proposed to encapsulate application, dependencies, and the required version of the *Cray Development Toolkit* (CDT) in a container image. This image would be used to build the application in a local machine. Due to the large size of CDT, however, they created a Python script to extract the application's binaries and their dependencies. These can be placed in a new lightweight container, which can be executed in a Cray HPC system, independent of which version of CPE is in the hosts. They proposed two use cases for this approach. The first is for research reproducibility, by enabling to keep a version of the application compiled with a specific software environment. The other is for testing the performance of an application with different CPE versions (e.g., to test performance regressions, or to run it with an older version of CPE that provides better performance). Another use-case they mentioned is testing new versions of the CPE before updating the system.

Researchers from LRZ (*Leibniz Research Center*) and Intel proposed using containers to deploy AI frameworks in a secure HPC environment [12], and presented their successful Charliecloud-based implementation on the SuperMUC-NG supercomputer. AI researchers and developers are used to configuring their environments by using tools that download the necessary software from the Internet. For security reasons, however, many HPC systems do not allow external network access. In addition, pre-configuring the nodes is often impossible, due to incompatible dependencies between different software, or even between different versions of the same software. Their proposal is to employ containers to create *user-defined software-stacks* (UDSS) [69]. Thus, allowing the user to pre-configure the environment, before transferring it to the HPC system. Using Charliecloud, they were able to set up a software stack, and to implement and test the full experimental pipeline in less than one day. Moreover, the authors highlight the importance of being able to use the Intel-optimized version of TensorFlow, and Intel MPI in order to achieve optimal performance.

Rudyy et al. [72] presented a study on performance and portability of containers on four different systems. They were able to deploy and execute a containerized

CFD application on the *Lenox* cluster (Intel Xeon based, 1 Gb/s Ethernet) using Docker, Shifter, and Singularity. All three used the same Docker image, the only difference being that for Singularity the image must first be converted using the *singularity-build* tool. On a second set of experiments, to analyze the portability of containers for HPC, they evaluated Singularity-based executions of the same CFD code on three systems with different architectures. These were: (a) the *MareNostrum4* supercomputer, a large-scale Intel Xeon based system with an 100 Gb/s Intel OmniPath interconnection network; (b) CTE Power, which is based on the IBM POWER9 processor and an InfiniBand network; and (c) ThunderX, an ARMv8 system whose nodes are interconnected via 40 Gb/s Ethernet. The containerized application was able to achieve near bare-metal performance by using Singularity's ability to access the host's MPI, thus being able to take advantage of the available high performance interconnects.

When creating three different Singularity images to evaluate the execution of real-world scientific applications on *Stampede 2*, Wang et al. [89] observed that the two images that used the same Linux distribution as the host were able to execute the applications right out of the box. On the other hand, the third image, which was from another distribution, needed minor modifications. Their overall conclusion, based also on their overhead evaluation, was that it is viable to provide customizable container images for HPC users while still providing optimal performance for their applications.

The *Argonne Leadership Computing Facility* (ALCF) has employed Singularity to provide portability of software stacks between systems, including development systems and large-scale supercomputers [29]. One of their stated goals is to make HPC more approachable, especially for new users and users from research fields which may not have a strong computing culture. Singularity was used to run *ATLAS* (from the *Large Hadron Collider's ATLAS experiment*) on an ALCF supercomputer, and to share *CANDLE* (*CANcer Distributed Learning Environment*) framework workflows among users, as well to enable these users to tailor its software stack to their needs.

Yuan et al. [94] presented a system for scientific collaboration on the *Coastal Resilience Collaboratory* (CRC), and to reduce the necessary effort to run scientific applications on different systems. They use Docker to deploy Jupyter notebooks to user machines. These are used by researchers and engineers as an interactive tool to customize their simulations. Data sharing and access to HPC resources is handled by the *Agave* framework, and Singularity is used to deploy the simulation images in cloud-enabled resources. Besides that, they created a *Coastal Model Repository*, which provides a collection of container images targeting cloud-like infrastructures.

To analyze the use of containers for the portability of HPC applications, Peiro Conde [67] built several images for different applications (including the OSU Microbenchmarks, OpenFOAM, and GROMACS), and evaluated these in an ARM-based cluster. Based on their experience, they suggest that system administrators provide Docker and Singularity usage guidelines to all users. And, moreover, that they provide base Docker images for their systems, which can be used by experienced users to create custom images. These images would then be shared in a repository, to be used by all users of the system. Among the potential flaws to this workflow, they

cite the large amount of work and time required to create the container images, and the potentially large storage space requirement for the custom images. As a final remark, Peiro Conde [67] state that “*there is still a lot of room for improvement in usage*” and that “*HPC-oriented containers such as Singularity require users with good expertise on the matter.*”

With the intention of providing maintainers an information base to make informed choices when containerizing their MPI application, Hursey [37] analyzed several challenges faced during build and execution time. They presented common solutions to these, discussing the advantages and disadvantages of each. Among the explored challenges are library compatibility issues at build time, and cross-version incompatibilities at run time (between container and host binaries). They also discussed the choice between mapping one container per process or mapping one container per node. In addition, they took into account how container isolation mechanisms (e.g., *cgroups*, *namespaces*, *capabilities*) can hinder the efficiency of MPI implementations. The challenge of providing system-optimized MPI inside a container was also discussed, including three alternatives to overcome it. Finally, the authors included a discussion on how the experience of running containers on traditional HPC environments can be translated to a container orchestration environment, especially Kubernetes.

Aoyama et al. [2] integrated HPCCM (HPC Container Maker) into MEGA-DOCK, a large scale protein docking application. They aimed at integrating the container deployment workflow across HPC systems with different specifications. The HPCCM framework provides easy configuration of containers for different systems, handling issues such as host-dependent libraries and MPI-ABI compatibility between host and container. This integration was tested in the *ABCI* and *Tsubame 3.0* HPC systems, with good performance results (see Sect. 9).

Prominence [47] is a platform that allows users to run jobs across different resources while enabling these jobs to access both software and data. It uses Docker container images that are executed with either Singularity or udocker. In this platform, jobs preferably run locally, but have the ability to burst onto non-local clouds including, by order of preference, national research clouds, EGI resources, and public clouds. According to the authors, *Prominence* was successfully tested with Openstack, Microsoft Azure, AWS and Google Cloud, in addition to Openstack and OpenNebula clouds in EGI FedCloud. The authors remarked the loss in portability when a container was optimized to get the best possible performance in a specific platform, especially in a case where compiler optimizations specific to a CPU architecture made it incompatible with some previously supported systems.

Cerin et al. [17] designed an approach for containerizing an HPC resource scheduler. More specifically, the scheduler daemons hosted in the managed HPC nodes are executed inside containers. Moreover, they use *cgroups* to limit the amount of resources assigned to each of these containers. This way, it becomes possible to employ these nodes to run other applications or services simultaneously with HPC applications. The rationale behind this is to take advantage of the fact that many HPC applications are not able to utilize the whole capacity of the computation nodes at all times. By isolating the resources available to the container manager, it becomes possible for more general purpose jobs to co-inhabit with the HPC jobs, allowing

them to execute concurrently while isolated inside their own containers. These non-HPC jobs would use the potentially underutilized resources in the node, therefore, increasing resource utilization. The implementation presented by Cerin et al. [17] is based on Kubernetes, to manage and orchestrate both the job manager (Slurm) containers and the general purpose ones. The container runtime is CRI-O, which is meant to be a lightweight alternative to Docker and is designed with privilege separation in mind. This work, however, does not present any evaluation of the proposed approach.

To get a deeper sense of the feasibility of using containers in HPC, Ruhela et al. [73] employed a different system and runtime to run the same container image from previous experiments (see Sects. 8.1, 9 and 8.5). Instead of the *Frontera* supercomputer and Singularity or Charliecloud, they employed a cluster that resembles the *Stampede2* HPC system, and used Podman. All workloads ran correctly, but there was a minor performance degradation (5–10%). The authors hypothesize this is because Podman uses *fuse-overlayfs*, and employs additional inter-process isolation compared to Singularity and Charliecloud.

Vaillancourt et al. [87] developed a system that automates tasks related to deploying scientific workloads in cloud environments. Its goal is to reduce the time needed to adapt existing HPC applications for running in cloud environments. It automatically manages the different deployment options of the available computing clouds, and configures networking support for parallel application execution. In addition, it can translate legacy application build and deployment mechanisms to their cloud equivalents. The authors highlight two compelling cases for moving to the cloud: (1) when computational resources are not available anywhere else, and (2) when cloud resources are more readily available than HPC-style resources. They also mention that “*conversely, the public cloud might be cost-prohibitive for very large-scale or large-output applications [...]*.” Their tool combines Docker, to provide portability and reproducibility, with Terraform and Ansible for automatic deployment, management and provisioning of cloud resources. They used it to encapsulate, deploy, and run three representative scientific application workflows (*Lake_Problem_DPS*, *WRF*, and *FRB_pipeline*) on AWS and on the *Aristotle Cloud Federation*.

The *EASEY* framework [34] aims to facilitate the deployment of applications on “*very-large scale*” systems, with minimal interaction. Its goal is to reduce the time scientists spend on deploying and tuning their application on new systems, as well as on job submission. They discuss some initial development, which enables the transformation of Docker-based applications into Charliecloud containers. Moreover, to optimize the performance on the target system, they have a mechanism that enables the automatic addition of system-specific libraries to the container. *EASEY* was evaluated using a Docker container image for the LULESH benchmark, which was ported to DASH. Charliecloud-based executions on up to 32,768 cores of the SuperMUC-NG supercomputer presented negligible overhead compared to native performance. In addition, *EASEY* reproduced its scaling behavior, and kept its performance very close to the original manually compiled and naively executed application.

Jung et al. [39] developed an infrastructure based on Docker and Kubernetes to facilitate the execution of the *ROMS* model (*Regional Ocean Modeling System*) in

different public (AWS, Google Cloud, and Microsoft Azure) and private (a laptop, and two different clusters) computational environments. They created a ROMS container image, which they executed in cloud clusters with different core counts. The authors claim that, in a variety of public and private environments, the container-based architecture makes the numerical ocean modeling much easier than the VM-based one.

Ruhela et al. [74] employed five different HPC systems to perform an evaluation of containers applied to medium and large-scale HPC workloads. Regarding container portability, they remark on the difficulty brought by working with different MPI implementations and CUDA versions. They mention situations where different containers had to be used for different experiments, and also for different clusters. The latter was due to each system requiring different MPI implementations, to achieve optimal performance. They also reported some difficulty on installing CUDA libraries in Charliecloud's unprivileged containers. In most cases, this was solved by the runtime's capability of injecting the host's CUDA libraries into the container, but they were still unable to run one of the applications in one system. They also ran into difficulties caused by the misunderstanding of requirements for containerized workloads or by misconfigured installations, which required help from system administrators to solve. Such issues result from cluster having different software ecosystems and from differences between how containers and traditional HPC applications work. Finally, they conclude that *"while in some cases containers provide portability, in large-scale multi-cluster situations, containerization does not simplify the work needed to complete the tasks and adds an additional level of complexity to the existing scientific applications."*

Chang et al. [18] analyzed the feasibility of running MPI-based applications encapsulated in Singularity containers on NASA's *High-End Computing Capability* (HECC) resources. HECC provides access to both in-house resources and commercial cloud resources from Amazon AWS. Their experiments comprised one HPC workload and two machine learning workloads. For the HPC tests, they employed JEDI Singularity containers with a *Finite-Volume Cubed-Sphere Dynamical Core* (FV3)-JEDI atmospheric simulation. This workload was tested on both in-house and AWS resources. The machine learning workloads were a recommender system called *Neural Collaborative Filtering*, and *Resnet-50* trained on the *ImageNet* data set. These were run inside a *TensorFlow* container from the *NVIDIA GPU Cloud* (NGC), and were tested only on in-house resources. The authors provide a detailed presentation of all the hurdles they faced when trying to run these workloads on different on-site HECC resources and on AWS resources. Running them on a single node, with a single container, did not present any significant difficulties, since access to the host's MPI is not necessary. Running on multiple nodes, however, presented a series of issues, especially due to requiring access the host's MPI. There were also difficulties due to differences in network configuration on different resource types, and on the AWS resources they had to replace their default shared file system for Lustre. The final conclusion by Chang et al. [18] was that while Singularity-based containers with hybrid container-side and host-side MPI is feasible, users are likely to run into issues whose solution require knowledge of Singularity,

MPI libraries, and of the network infrastructures. Therefore, most users would need some help to run their workloads, as such issues are usually handled by HPC-site support staff. While this work includes some container performance measurements, they were not compared with bare-metal performance.

Brayford et al. [13] demonstrated the building, configuration, and deployment of a quantum gate simulation application developed in *Julia* and containerized with Charliecloud on multiple HPC systems with different CPU architectures (*Arm ThunderX2*, *Arm Fujitsu A64fx*, *AMD Rome* and *Intel Skylake*) and instruction sets. Moreover, they were able to combine the software environment inside the container with host-side software, to be able to use a tool that was installed in the hosts to profile the application installed inside the container. They considered the results extremely promising and that they justify the effort of developing the HPC-specific containers.

Zhou et al. [98] proposed a dual-level scheduling strategy for container jobs in HPC systems, by creating a bridge between an HPC workload manager and a container orchestrator. It is aimed at being used in a hybrid architecture composed of an *HPC cluster* and a *Cloud cluster*. To accomplish this they implemented a tool named *TORQUE Operator* which bridges the *TORQUE* HPC workload manager and Kubernetes. *Kubernetes* schedules jobs on the first level, and on the second level they are scheduled by *TORQUE*. In addition, their chosen container was Singularity. Its key architectural aspect is that the login node for *TORQUE*, which submits the jobs on the HPC cluster, is a VM that serves as a Kubernetes worker node. Moreover, *TORQUE Operator* also includes user management logic, to ensure that HPC cluster jobs belong to the correct users, thus assuring confidentiality and the right user privileges. Besides presenting their architecture and tool, Zhou et al. [98] present two agriculture-related machine learning use cases, and a performance evaluation using the BPMF MPI benchmark. All three applications show performance gains by running in the HPC cluster compared to cloud VM, and two of them scale up to 20 nodes (one of the ML codes is sequential). Finally, this work does not discuss or evaluate containerization overheads.

BeeSwarm [85] is a system that can be integrated with existing continuous integration (CI) systems to perform performance and parallel scaling tests for HPC applications. It is based on a modified version of BEE [19], which was presented earlier in this section. During CI testing, the tool builds a container for the tested application and triggers its deployment in the platform where the scalability tests will be executed. *BeeSwarm* is responsible for allocating the computational resources for the scalability tests via backends that support different cloud APIs, as well as HPC batch schedulers. The test output is sent back to the CI environment, where it is parsed to generate a report for the user. They present use cases employing *Travis CI* and *GitLab CI* using Docker containers to deploy the tests into resources of the Chameleon Cloud, as well as employing *GitHub Actions* using Charliecloud containers to deploy the tests on the *Google Compute Engine* (GCE).

Tippit et al. [83] report on their team's use of Singularity to distribute their GPU-based quantum simulation software developed in the *Julia* programming language. They remark on how this allows them to share their work with others while ensuring that software versions and libraries will exactly match their own. Moreover, they

mention the advantage brought by the reduced performance overhead of containers compared with hypervisor-based VMs.

The *Physikalisches Institut University of Bonn* has employed containerization on a hybrid HTC and HPC cluster, to support the specific requirements of CERN's ATLAS experiment software stack, while also allowing individual users to choose the OS where their jobs should run [23]. Moreover, they say that implementing this setup was made straightforward by the container awareness of the HTCondor resource management system. In this scheme, HTCondor is responsible for instantiating the container corresponding to the user specified OS and for making sure that users never access the bare-metal machine. One difficulty they had was that supporting interactive jobs on containers instantiated by HTCondor required some "tricky" SSH configuration. In their system, users have access to container images based on official Linux distribution images, with some adaptations. These are automatically rebuilt daily or when their recipes are updated. They chose Singularity as their container runtime, due to off-the-shelf support by HTCondor, but mention plans to switch to a fully unprivileged solution.

Among other benefits, Kadri et al. [40] discuss containers as a solution to improve the portability of bioinformatics software.

The 41 works we analyzed in this section focus on different levels of portability enhancement that can be provided by containers. The most basic level consists of facilitating the setup of user-defined software stacks. On the next level, we have works aimed at easing the portability of applications between different dedicated HPC systems, or different cloud-computing platforms. Finally, we have works on providing portability between different types of systems, which may include development systems, dedicated HPC systems, or cloud-computing environments. The former may be useful for setting up a CI/CD pipeline, or to implement an automated application (or experiment) configuration-execution workflow. Besides these, there are also a few works aimed at facilitating the distribution of a single application, or at providing base container images to provide pre-configured environments aimed at a specific field (such as AI), or at providing optimized environments for specific systems. Finally, we have a few works that analyze the portability of HPC application containers.

Among the analyzed works, we are most interested on the ones that propose solutions to provided portability between different systems, especially those aimed at HPC and cloud-computing platforms. Overall, including those focused on a single system, more than three quarters of the works in this section deal with dedicated HPC systems, while those that deal with cloud environments amount to only slightly over one fifth of them. Moreover, if we restrict ourselves to works that allow portability between different systems, close to 36% of the 41 works deal with portability between different HPC systems, about 7% between cloud platforms, near 21% between dedicated HPC systems and cloud, and about 12% between personal (or development) computers and the cloud. This could indicate some potential for research, especially on improving the portability of applications between dedicated HPC systems and cloud resources.

12 Research reproducibility

The portability of containers allows the same software environment to be used to repeat experiments on different systems, by different researchers. Therefore, containers can provide enhanced reproducibility of execution for scientific application. This is an important aspect in attaining reproducibility of research results. Especially in view of the large array of scientific and engineering fields that rely on computational data processing and simulation. In this subsection, we present the contributions of the surveyed works to research reproducibility.

Singularity Hub is a container repository and a framework to build and deploy Singularity containers [79]. It aims to make containers movable and transparent, while enabling customizable builds and scalable deployment. It also includes tools to facilitate the comparison and assessment of the reproducibility of containers. According to Sochat et al. [79], *"this open standard and these functions support the larger underlying goal of encouraging data sharing and reproducible science."* Besides Singularity Hub, this work presents novel reproducibility metrics to compare containers based on a desired level of reproducibility. The authors present a set of pre-specified reproducibility levels, including *"identical"*, *"replicate"* (built in different machines or at different times), *"recipe"* (different environments, labels, and runscripts), and *"base"* (same base container).

Hisle et al. [33] worked on the containerization of *Autodock Vina*, a molecular docking program used in the search for potential drugs among a large set of molecules. The authors explain that this program is known to run significantly faster when compiled for the target hardware, compared to the available pre-compiled binary. Moreover, they remarked that speeding-up Autodock Vina should result in important savings in HPC resource usage, since the docking process often requires hundreds of thousands of CPU-hours. One problem, however, is that its setup and compilation is time-consuming and difficult, due to dependencies on outdated compilers and libraries. By using Singularity to containerize the application and its dependencies, the authors aim to enable users to easily compile it in their target systems. Moreover, they observe that most of its users lack the knowledge and skills to manually set up such an environment, thus elevating the importance of the pre-configured container. Results from this work show that, compared to the pre-compiled binaries, the execution time of the source-compiled binaries is 33% shorter.

Martinasso et al. [57] demonstrated how they can use containers to ensure experiment reproducibility even after the HPC system's execution environment is updated. They specifically focused on supporting applications built with different versions of the *Cray Processing Environment* (CPE). As a use case, first they compiled the *COSMO* weather forecast model inside a container containing a specific version of CPE. Then, they used a script to extract the generated binaries and dependencies, which they placed in a new lightweight container image that enables executing the application in the HPC system, independent of the version of CPE installed in the host. Therefore, this eliminates the need for the weather researchers to rebuild

COSMO at every CPE update, which also entails repeating previous experiments to ensure numerical reproducibility.

Heinonen [29] reports on the benefits of using Singularity for research at the *Argonne Leadership Computing Facility* (ALCF). The author states that reproducibility “*can be seen as one of the primary assets of container technology*”, due to its crucial role in scientific research.

The *Cloud Resilience Collaboratoty* (CRC) developed a container-based system to enable and facilitate the collaboration between its members [94]. Simulations are configured through Jupyter notebooks encapsulated in Docker containers, and deployed in the cloud as Singularity containers. The authors argue that both beginner and computer-literate users can benefit from such a system, which not only makes all machines look the same for the user, but also encapsulates and tracks the provenance information of the experiments, thus leading to better reproducibility.

Vaillancourt et al. [86] developed a method to create infrastructure-independent reproducible containers for scientific software. Their solution is based on providing container templates based on best practices for each environment. It was implemented using Singularity and the *Nix* package manager. *Nix* allows to specify the specific versions of packages and dependencies to be installed. The container templates, which are stored in a *Container Template Library* (CTL), consist of a Docker file along with necessary scripts, including a *Nix* configuration file. This helps to ensure the reproducibility of the container, as long as *Nix* is supported in the target environment. This work, in specific, includes the provision of container templates that can run in various cloud environments. Moreover, the authors demonstrate a “*flexible HPC-style infrastructure that is deployable across multiple cloud providers*.” They employed this infrastructure to execute HPL on virtual clusters in two different cloud infrastructures (JetStream and Red Cloud).

Cavet et al. [16] reported on the progress of the ComputeOps project, which aims to “*explore the benefits of containerization for the IN2P3* (Institut national de physique nucléaire et de physique des particules) *scientific communities*.” Among other topics, the authors highlight the usability of containers in continuous integration pipelines. Through empirical evaluation, they defined that this is the best solution to fulfill their objectives. At the time this work was published, they were working on a *Comprehensive Software Archive Network* (CSAN), aimed at enabling the full reproducibility of containers. CSAN builds upon Singularity and the *Guix* package-manager.

Besides portability, the container image and infrastructure created by Jung et al. [39] to run ROMS also aims at improving reproducibility. The results of this containerized model, executed in various container clusters with different node counts, were the same from their control model, regardless of OS or hardware environment. Citing the authors, “*This suggests that a container-based cluster is appropriate for use in the computational reproducibility of the numerical ocean model*.”

Kadri et al. [40] remark on sharing bioinformatics software as a container as a means to achieve reproducible behavior, since “*the contents, including the version of the software and its dependencies, are locked down*.”

While it does not seem as popular a research topic as the portability that supports it, several works have investigated the contribution of container environments to reproducibility. This being one of the main contributions of this technology to scientific research and, specifically, to scientific computing [29]. The works above touch into providing general support for (a) building, storing, and sharing reproducible containers [40, 79]; (b) quantifying such reproducibility [79]; (c) improving the reproducibility of specific applications [33, 39, 57]; (d) keeping results reproducible by allowing to execute different versions of a software environment on the same system [57]; (e) facilitating the sharing, configuration, and execution of a scientific application environment [94]; (f) providing a pre-configured and reproducible application environment that can be built in different systems [16, 86]; as well as (g) implementing infrastructure-independent reproducibility [86].

13 Unprivileged user containers

Unprivileged container execution, i.e., without administrative privileges, is of paramount importance in order to allow their usage in multi-tenant HPC systems. This requirement is pointed out by several authors [6, 8, 9, 12, 22, 26, 27, 31, 32, 35, 40, 48, 58, 62, 68, 72, 75–77, 88, 89], and is one of the main reasons why Docker did not gain as much traction in HPC as in other fields. Below, we summarize a few works that proposed solutions to this problem.

Higgins et al. [32] developed a *Docker API proxy* that acts as an intermediary between the Docker clients and the actual Docker daemon. To enforce security rules, it manipulates the content of client requests and of the responses to the caller. These rules consist of filters specifying keys that should be removed from client requests, and key-value pairs that should be merged into them. This proxy is able to identify the user that is communicating with it, and associates user information with their stored containers. Such information is used to ensure that the target container of received API calls is owned by the issuing user. Besides that, it is also used to filter the content of container listings, leaving only those that are actually owned by the requesting user. The authors say these strategies were already in place in the *Eridani* cluster at the *University of Huddersfield*. According to them, “they are effective to mitigate the privilege escalation and resource exhaustion attacks [...]”, and “this solution required no additional setup on the user’s behalf and operates transparently [...]”, thus allowing “[...] easy integration into the HPC environment without any significant configuration changes.”

Socker [4] is a Docker wrapper that aims to securely execute Docker containers on HPC systems, while respecting resource usage restrictions imposed by the job scheduler. To achieve secure execution, it runs the container as the user who submitted the job. To respect resource usage constraints, the wrapper enforces the membership of the container process in the cgroups the resource scheduler created for the job. One important limitation is that this approach only works if the container spawns only one process. Tests of *Socker* using MPI jobs submitted through Slurm demonstrated no performance overhead compared to Docker.

Gomes et al. [26] proposed *udocker*, an alternative Linux container implementation that enables the pulling, extraction, and execution of Docker containers in user mode, with no need for administrative privileges. In addition, *udocker* is a self-installing tool, whose installation can be done by unprivileged users. It assumes that container images will be prepared using Docker, on the user's personal computer. *udocker* implements four different container execution modes, which allows it to adapt to different host capabilities. These modes are denominated P, F, R, and S, with a few variations for modes P and F: (1) P modes are based on *ptrace* and implemented using *PRoot*; (2) F modes are based on *LD_PRELOAD* and implemented using *fakechroot*; (3) the R1 mode uses unprivileged user namespaces and is implemented using *runC*; and (4) the S1 mode is based on Singularity.

Sparks [81] proposed extending Docker's authorization framework to control access to various commands and the usage of runtime options. Their solution uses a Docker authentication plugin to allow configuring the Docker daemon for granular access policies. Besides that, the daemon can also be configured to leverage extended authentication mechanisms. The system administrator would be the one responsible for registering plugins as part of the configuration and startup of the Docker daemon. Unfortunately, the authors did not present any evaluation or real-world use case of this scheme.

Sarus [9] is an OCI-compliant container implementation that extends *runC* to address unique requirements of HPC environments. These include obtaining native performance from dedicated hardware, improved security on multi-tenant systems, support for network parallel file systems and diskless nodes, and compatibility with workload managers. *Sarus* is composed of a CLI (*command-line interface*) component, an image manager, and a runtime component. Container processes are owned by the user who executed the container launching command, thus avoiding privilege escalation. Additionally, *Sarus* is capable of swapping the MPI runtime inside the container with an MPICH-ABI compatible implementation from the host. Besides, it is able to use the *NVIDIA Container Runtime* hook, to enable accessing GPUs inside the containers.

Priedhorsky et al. [70] argue that the difficulty of building container images with unprivileged tools is slowing down the adoption of containers by supercomputer users, even when the supercomputers themselves support containers. With this in mind, they explore the issue of container build privilege in HPC, including the different container privilege models, and what makes building containers in an unprivileged environment harder than running them (e.g., package managers assuming privileged access, with packages requiring multiple UIDs and GIDs, and the execution of privileged system calls on installation). Moreover, they analyze the strategies adopted by *Rootless Podman* and *Charliecloud* to solve this problem. *Rootless Podman* utilizes a mount namespace with a privileged user namespace, employing multiple container GIDs and UIDs which are independent of the host's. The root user inside the container is mapped to an unprivileged host user. This model requires privileged host tools to assist building, and computer sites must maintain ID mapping files to ensure proper isolation between different users. In addition, according to Priedhorsky et al. [70], *Podman*'s namespace implementation was incompatible

with shared file systems. Charliecloud, on the other hand, uses a mount namespace and an unprivileged user namespace, with only one UID and GID mapped into the container (aliases to the user's IDs in the host). This model enables fully unprivileged container building and execution. The main advantage is that it does not rely on the correct configuration of the helper tools, like maintaining proper the user ID maps. This is accomplished by automatically injecting the command line tool `fak-eroot` into the building process. This tool acts as a wrapper that fools the wrapped process into thinking it is running with root privileges.

LPMX [91] is an unprivileged container implementation aimed at providing mutual composability to containers, especially for use in scientific pipelines, such as the ones commonly employed in genome analysis. More specifically, it allows encapsulating the different tools that compose the pipeline into separate containers while retaining their ability of calling other tools that are outside the container (i.e., in the host or in another container). Moreover, they employ a layered file system to reduce the size of the container. Different from other implementations, however, this file system runs completely in user space, thus not requiring administrative privileges. Finally, just like udocker, it does not require administrative privileges neither to build nor to run containers, thus being fully rootless. In tests, it displayed considerable software installation and compilation overheads, but minimal overheads for genomic analysis application execution.

As we have seen, several works tried to solve the problem imposed by privileged container execution. Some of them, by modifying Docker or implementing extra measures to make it more secure [4, 32, 81]. A few others, by proposing novel container technologies to allow the unprivileged execution of containers [9, 26, 91]. Besides these, we have the HPC container implementations seen in Sect. 4, which also implement unprivileged execution. Some of these still require administrative privileges for building the containers, but there are already solutions that managed to eliminate this requirement.

14 Related work

Pahl et al. [66] published a state-of-the-art review about cloud container technologies, in the form of a systematic mapping of 46 papers published between 2007 and 2016. Interesting observations from this review include that (1) *"there is more work on deployment and management services than design and architecture concerns"*; (2) there is an equal spread between IaaS and PaaS focus; and (3) evaluation is mainly done through experiments and case studies. The authors conclude that *"container technology is still in its formative stages"*, and that there is a lack of tools to automate and facilitate container management and orchestration. It is worth noticing that, despite being published in 2019, *this paper was actually submitted in 2017*. The authors also detected a growing interest and usage of containers as lightweight virtualization solutions at the IaaS level, and as application management solutions at the PaaS level. Finally, they indicate the possibility of using containers to support continuous development in the cloud. Among other analyses, they looked at the distribution of studies by *community*, by *motivation*, and by employed *container*

technologies. Their results show that almost half of the reviewed works were developed by the *Cloud* and *Big Data* community. At second place was the distributed-systems community (23% of the studies). Noticeable is the absence of works by the *HPC community*. Pahl et al. [66] also showed that the main motivation for using containers, in the analyzed sample, was *lightweight resource management* (33% of the works). The second most frequent was *ease of deployment* (26%), followed by automatic deployment (15%), and large-scale deployment (9%). Additionally, the main container technology employed by the works was Docker (40% of the studies), followed by LXC (21%). Notable is the absence of HPC targeted container technologies, such as Singularity, Charliecloud and Shifter. Possibly, because this survey only includes works published before these technologies became available.

Medrano-Jaimes et al. [58] published a technology review of container platforms designed for HPC, based on a collection of 10 works published from 2015 to 2018. They discuss containers as a solution to allow user environments in HPC systems, in contrast to traditional approaches, such as Linux modules, package managers, and *Python* virtual environments. According to them, these approaches do not fix the issue of portability, making it difficult for users to transfer their work to a new system. Moreover, they remark on the use of containers in science, to provide mobility and reproducibility, in addition to portability. This work includes an analysis of three HPC container technologies, namely Charliecloud, Shifter, and Singularity. The authors considered Charliecloud the easiest to use, due to demanding little effort from users and sysadmins, and providing simple and straightforward commands. Shifter was remarked for being a proven solution for working with Docker containers, but with the drawback of its installation and maintenance processes requiring more commitment from the system administrators. Regarding Singularity, they identify the native InfiniBand support as its main differential. They also mention the capability of configuring its software according to site policies, and the existence of a cloud service for storing and sharing Singularity containers. The potential performance improvement when using Shifter or Singularity to execute applications that are not optimized for distributed file systems is also remarked upon. Among their conclusions, they say that the container solutions for HPC environments present, in general, satisfactory results. Nevertheless, they argue that "*there is still a lot of work to be done*". Finally, they highlight the continuous growth of the use of containers for HPC, attributed to advantages like the improved scientific reproducibility.

Bachiega et al. [5] published a survey on containers, with focus on performance evaluation. It comprehends 25 works published from 2010 to 2017, and is based on a systematic mapping considering main databases like *Springer*, *IEEE*, *ACM*, and *Scopus*. The selected papers were classified according to the performance evaluation method, among *analytic performance modeling*, *simulation performance modeling*, or *performance measurement*. The authors found out that the *performance measurement* technique was extensively used, in different scenarios. The 22 works that employed this technique, including several HPC-related ones, utilized benchmarks scientific applications, and other virtualization tools. The authors remark, however, on "*the lack of comparison of containers with the use of public networks, data migration, disaster recovery, and parallel programs*." Moreover, they found only one work that employed analytic modeling, and it was restricted to a specific

container environment. Besides, they found only two papers using the simulation technique, both describing *ContainerCloudSim*.

To analyze the state-of-the-art in container technologies, Casalicchio and Iannucci [15] surveyed 97 works published until 2018. Interestingly, 90% of these were published between 2015 and 2018, and 50% of them in 2017 and 2018. Their goal was to identify the most demanding application fields employing containers, the related challenges, the proposed solutions, and the remaining open issues. Among the application fields, the majority of the analyzed works is focused on orchestration (38%), followed by applications (26%), security (23%), and performance comparison (13%). In high performance computing, which they consider a subcategory of applications, they identified the demand for ease of deployment of user-defined software stacks or virtual HPC clusters, and for leveraging the flexibility of containers in real life computation and data intensive jobs. Additionally, they identified four challenges related to the use of containers applied to HPC: (1) container support for MPI and CUDA and for execution on GPU and FPGA; (2) the scheduling of containerized HPC jobs; (3) mechanisms to guarantee native performance in containers; and (4) solutions to guarantee the proper level of security in terms of isolation, data security, and network security. They also remark on the HPC-specific challenges of efficiently scheduling containerized IO-intensive jobs, and mitigating or solving the IO performance limitation of containers. The authors conclude that more effort should be applied to solve challenges in IO performance, multilayer adaptation, energy efficiency, and security. Moreover, they remark that security is probably the challenge with most urgent needs. The only HPC focused container technology to appear in this survey is Charliecloud [69], even though technologies like Shifter and Singularity already existed in the period it comprehends.

Bentaleb et al. [10] published an extensive overview of virtualization focused on containers. They summarize important aspects of container technology, including container implementations (Docker, Singularity and udocker are discussed in more detail), container architecture, and container lifecycle, and container orchestration. Then, they discuss taxonomies of container technologies found in the scientific literature and propose “*a new one that covers and completes those proposed in the literature.*” Following that, they discuss applications of containerization in different application domains, including *big data*, *high-performance computing*, *grid computing*, *high-throughput computing*, *cloud computing*, *internet of things*, *fog and edge computing*, and *DevOps*. There is a short subsection for each domain, and the one about HPC only provides a superficial view of the application of containers in this domain, comprising only 7 related works. Finally, this work also provides a quick overview of the metrics that are commonly used to evaluate containers.

Only two of the five related works above are specific to HPC [5, 58]. The most up-to-date survey [15], to the best of our knowledge, only includes works published up to 2018. In addition, while the overview of container technology published by Bentaleb et al. [10] is more recent, it only includes a short subsection on HPC, where they only cite seven references. In comparison, our survey is specific to the HPC domain, is more up-to-date than the surveys above, and comprehends a substantially larger selection of works related to containers in HPC than the overview paper.

15 Conclusion

This work is a research literature survey on the use of containers for high performance computing (HPC). We carefully selected 93 papers published between 2016 and mid-August 2022. The last survey we could find on the subject only analyzed papers published up until 2018, with the vast majority of them being published before or in 2007. There is an overview of container technology that is more recent than these surveys [10], but it provides a very superficial view of their application in HPC, in a short section that only discusses seven related works. So, we believe this survey provides an important, and necessary, updated overview of the progress of research on the subject.

We can clearly see that most works are focused on analyzing or mitigating potential performance overheads due to containerization. These studies are normally empirical, and compare the performance of benchmarks and applications running inside containers, with their performance in other environments, such as virtual machines, or directly on the host's native environment (bare-metal). Their results demonstrate that, in most cases, containers do not incur significant computational performance overhead compared to native execution. Moreover, in this aspect, containers are generally better than VMs.

In terms of network performance, however, containers may be significantly slower than native execution. But this depends on the capabilities and configuration of the container runtime and image. There are, however, a series of techniques that allow containers to overcome this limitation, mostly dealing with giving the container access to the host network. In the case of HPC, there is an extra difficulty related to giving software inside the container access to specialized network hardware, which may need specific drivers or libraries to be operated. One way to work around this is to mount-bind the required software, including host-optimized MPI implementations, into a directory inside the container. It is possible to link the user application to these libraries, thus getting access to the host's specialized hardware. A similar approach would be to simply copy these files inside the container. This technique can also be used to give containerized applications access to GPU accelerators. One main drawback of these approaches is that they break portability, since they are very system-specific. Another approach is to rely on *Application Binary Interface* (ABI) compatibility between the host's MPI implementation and the one inside the container. For example, if both the host and the container MPI have compatible versions of the MPICH-ABI, then it is possible for the containerized application to use the host's MPI, which allows access to the host dedicated HPC interconnect. A secondary concern for all the techniques presented here is that they reduce container isolation. But this is not a major concern for HPC, since it is not usual for more than one user to be allocated on the same host, and security concerns are managed at host-level. Moreover, we could argue that it makes no sense, in most cases, for a user to instantiate more than one container per host.

In terms of storage performance (IO), there seems to be some degradation, especially when reading and writing data to Docker's layered file system, which should

be avoided. In other cases, unless there are multiple containers per host, the storage performance should be close to native.

Considering the above, we can conclude that, in terms of performance overhead, containers are a good solution for HPC, especially compared to hypervisor-based VMs. Moreover, getting near-native computing performance with containers is straightforward, and the other kinds of overhead, such as communication, have solutions that are already supported by HPC-specific container implementations.

In what concerns software portability, containers are clearly a very useful tool. They provide users the benefits of virtualization, without the need to configure and deploy a whole new system, as required by traditional VMs. This, and the reliance on publicly available base images, makes for a considerable reduction in the size of the customized environments, making them easier to share and faster to deploy on new systems. Another clear benefit of building upon a base container is the possibility to use pre-configured containers which already include commonly used software environments. These can be found on public container repositories or provided by system administrators. Finally, an important consequence of container portability, is making it easier to provide reproducible applications, which is of utmost importance for science.

One surprising aspect is that, while there are plenty of works dealing with portability, few of them [18, 19, 39, 47, 72, 87, 93, 98] deal with the portability of HPC applications from dedicated HPC systems to cloud environments. Most works deal with portability between different users, different HPC systems, and from personal computer to HPC-systems. Works that deal with cloud environments seem to mostly focus on the portability of traditional cloud workloads from the cloud to HPC systems (e.g., machine-learning and big-data applications). There also seems to be a prevalence of the usage of private clouds in these works. This points to an unexplored research opportunity on facilitating the portability of traditional HPC workloads (usually executed on dedicated HPC systems) to the cloud. This could be an interesting topic to explore, specifically focusing on the portability between the user's personal computer, HPC systems, and public cloud environments.

In terms of container implementations, our data demonstrates a clear growth of interest in HPC-specific runtimes, which seems to follow the general growth of interest in using of containers for HPC. Runtimes such as Shifter, Singularity, and Charliecloud are already able to deal with most of the specific challenges and desired aspects we presented in Sect. 3. Besides, this seems to be the direction the HPC research community has favored, rather than relying on more standard solutions, such as Docker.

As future work, we are interested in developing techniques and tools to ease the migration of traditional HPC workloads to public cloud environments. This move to the cloud has the potential to reduce the barrier to adoption of HPC by projects and institutions that do not have access to, and cannot afford to own, a dedicated HPC system.

Appendix

Useful and commonly repeated acronyms and abbreviations: ABI: Application Binary Interface; AI: Artificial Intelligence; AMG: Algebraic Multigrid method; API: Application Programming Interface; AUFS: Docker's Union File System; AWS: Amazon Web Services; BEE: Build and Execution Environment; CCE: Cray Computing Environment; CDT: Cray Development Toolkit; CFD: Computational Fluid-Dynamics; CFQ: Completely Fair Queuing; CI: Continuous Integration; CI/CD: Continuous Integration and Continuous Delivery; CLE: Cray Linux Environment; CLI: Command-Line Interface; CMA: Cross Memory Attach; CPE: Cray Processing Environment; CPU: Central Processing Unit; CV: Computer Vision; DASH: a C++ template library for distributed data structures; DFE: Data-Flow Engine; EC2: [Amazon] Elastic Compute Cloud; ELF: Executable and Linkable Format; FPGA: Field-Programmable Gate Array; GID: Group Identification; GPU: Graphics Processing Unit; HPC: High Performance Computing; HPCC: High Performance Computing Challenge; HPL: High Performance LINPACK; HTC: High Throughput Computing; IB: InfiniBand; IMB: Intel MPI Benchmarks; IO: Input and Output; IOR: a parallel IO benchmark; IP: Internet Protocol; IPC: Inter-Process Communication; IPoIB: IP over InfiniBand; IT: Information Technology; KMI: K-mer Matching Interface; KNL: [Intel] Knights Landing; KVM: Kernel Virtual Machine; LXC: Linux Containers; LXD: An interface for LXC; MCDRAM: Multi-Channel Dynamic Random Access Memory; MD: Molecular Dynamics; ML: Machine Learning; MPI: Message Passing Interface; NAS: NASA Advanced Supercomputing Division; NFS: Network File System; NIC: Network Interface Controller; NPB: NAS Parallel Benchmark; NUMA: Non-Uniform Memory Access; OCI: Open Container Initiative; OS: Operating System; OSU: Ohio State University; PCI: Peripheral Component Interconnect; PFS: Parallel File System; PID: Process Identification; RAM: Random-Access Memory; RDMA: Remote Direct Memory Access; RNN: Recursive Neural Network; ROMS: Regional Ocean Modeling System; SDN: Software-Defined Network; SHM: Shared Memory; SMT: Simultaneous Multi-Threading; SR-IOV: Single Root IO Virtualization; SSH: Secure Shell; TCP: Transmission Control Protocol; TF: TensorFlow; UDI: User-Defined Image; UDSS: User-Defined Software Stack; UID: User Identification; VCC: Virtual Container Cluster; VM: Virtual Machine; VXLAN: Virtual Extensible Local Area Network; WRF: Weather Research and Forecasting [model].

Funding This work received partial financial support from Petrobras, FAPESP (Grants 2013/08293-7 and 2019/12792-5), and CNPq (Grant 314645/2020-9).

Data availability Data sharing is not applicable to this article as no datasets were generated or analyzed during the current study.

Declarations

Conflict of interest The authors have no relevant financial or non-financial conflicts of interest to disclose.

References

1. Abraham S, Paul AK, Khan RIS, Butt AR (2020) On the use of containers in high performance computing environments. In: IEEE 13th International Conference on Cloud Computing (CLOUD). IEEE, Beijing, China, pp 284–293. <https://doi.org/10.1109/cloud49709.2020.00048>
2. Aoyama K, Watanabe H, Ohue M, Akiyama Y (2020) Multiple HPC environments-aware container image configuration workflow for large-scale all-to-all protein–protein docking calculations. In: Supercomputing Frontiers. Springer, Cham, pp 23–39. https://doi.org/10.1007/978-3-030-48842-0_2
3. Arango C, Dernat R, Sanabria J (2017) Performance evaluation of container-based virtualization for high performance computing environments. [arXiv:1709.10140](https://arxiv.org/abs/1709.10140). Accessed 13 Sept 2022
4. Azab A (2017) Enabling Docker containers for high-performance and many-task computing. In: IEEE International Conference on Cloud Engineering, pp 279–285. IEEE, Vancouver, Canada. <https://doi.org/10.1109/ic2e.2017.52>
5. Bachiega NG, Souza PSL, Bruschi SM, de Souza SRS (2018) Container-based performance evaluation: a survey and challenges. In: IEEE International Conference on Cloud Engineering (IC2E). IEEE, Vancouver, Canada, pp 398–403. <https://doi.org/10.1109/ic2e.2018.00075>
6. Bahls D (2016) Evaluating shifter for HPC applications. In: Cray User Group, CUG, London, UK. https://cug.org/proceedings/cug2016_proceedings/includes/files/pap135s2-file1.pdf. Accessed 13 Sept 2022
7. Belkin M, Haas R, Arnold GW, Leong HW, Huerta EA, Lesny D, Neubauer M (2018) Container solutions for HPC systems. In: Proceedings of the Practice and Experience on Advanced Research Computing. ACM, New York, NY, USA. <https://doi.org/10.1145/3219104.3219145>
8. Beltre AM, Saha P, Govindaraju M, Younge A, Grant RE (2019) Enabling HPC workloads on cloud infrastructure using kubernetes container orchestration mechanisms. In: IEEE/ACM International Workshop on Containers and New Orchestration Paradigms for Isolated Environments in HPC (CANOPIE-HPC). IEEE, Denver, CO, USA, pp 11–20. <https://doi.org/10.1109/canopie-hpc49598.2019.00007>
9. Benedicic L, Cruz FA, Madonna A, Mariotti K (2019) Sarus: highly scalable Docker containers for HPC systems. In: International Conference on High Performance Computing. LNCS—ISC-HPC. Springer, Cham, pp 46–60. https://doi.org/10.1007/978-3-030-34356-9_5
10. Benteleb O, Belloum AS, Sebaa A, El-Maouhab A (2022) Containerization technologies: taxonomies, applications and challenges. *J Supercomput* 78(1):1144–1181. <https://doi.org/10.1007/s11227-021-03914-1>
11. Beserra D, Moreno ED, Endo PT, Barreto J (2016) Performance evaluation of a lightweight virtualization solution for HPC I/O scenarios. In: IEEE International Conference on Systems, Man, and Cybernetics (SMC). IEEE, Melbourne, Australia, pp 004681–004686. <https://doi.org/10.1109/smc.2016.7844970>
12. Brayford D, Vallecorsa S, Atanasov A, Baruffa F, Riviera W (2019) Deploying AI frameworks on secure HPC systems with containers. In: IEEE High Performance Extreme Computing Conference (HPEC). IEEE, Waltham, MA, USA, pp 1–6. <https://doi.org/10.1109/hpec.2019.8916576>
13. Brayford D, Allalen M, Iapichino L, Brennan J, Moran N, Q’Riordan LJ, Hanley K (2021) Deploying containerized QuanEX quantum simulation software on HPC systems. In: 3rd International Workshop on Containers and New Orchestration Paradigms for Isolated Environments in HPC (CANOPIE-HPC), pp 1–9. <https://doi.org/10.1109/CANOPIEHPC54579.2021.00005>
14. Canon RS, Younge A (2019) A case for portability and reproducibility of HPC containers. In: IEEE/ACM International Workshop on Containers and New Orchestration Paradigms for Isolated Environments in HPC (CANOPIE-HPC). IEEE, Denver, CO, USA, pp 49–54. <https://doi.org/10.1109/canopie-hpc49598.2019.00012>
15. Casalicchio E, Iannucci S (2020) The state-of-the-art in container technologies: application, orchestration and security. *Concurr Comput* 32(17):5668. <https://doi.org/10.1002/cpe.5668>. [e5668cpe.5668](https://doi.org/10.1002/cpe.5668)
16. Cavet C, Souchal M, Gadrat S, Grasseau G, Satirana A, Bailly-Reyre A, Dadoun O, Mendoza V, Chamont D, Marchal-Duval G, Medernach E, Pansanel J (2020) ComputeOps: container for high performance computing. *EPJ Web Conf* 245:07006. <https://doi.org/10.1051/epjconf/202024507006>

17. Cerin C, Greneche N, Menouer T (2020) Towards pervasive containerization of HPC job schedulers. In: International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD). IEEE, Porto, Portugal, pp 281–288. <https://doi.org/10.1109/sbac-pad49847.2020.00046>
18. Chang Y-TS, Heistand S, Hood R, Jin H (2021) Feasibility of running singularity containers with hybrid MPI on NASA high-end computing resources. In: 3rd International Workshop on Containers and New Orchestration Paradigms for Isolated Environments in HPC (CANOPIE-HPC), pp 17–28. <https://doi.org/10.1109/CANOPIEHPC54579.2021.00007>
19. Chen J, Guan Q, Liang X, Bryant P, Grubel P, McPherson A, Lo L-T, Randles T, Chen Z, Ahrens JP (2018) Build and execution environment (BEE): an encapsulated environment enabling HPC applications running everywhere. In: IEEE International Conference on Big Data (Big Data). IEEE, Seattle, WA, USA, pp 1737–1746. <https://doi.org/10.1109/bigdata.2018.8622572>
20. Chung MT, Le A, Quang-Hung N, Nguyen D-D, Thoai N (2016) Provision of Docker and InfiniBand in high performance computing. In: International Conference on Advanced Computing and Applications (ACOMP). IEEE, Can Tho City, Vietnam, pp 127–134. <https://doi.org/10.1109/acomp.2016.027>
21. Chung MT, Quang-Hung N, Nguyen M-T, Thoai N (2016) Using Docker in high performance computing applications. In: IEEE Sixth International Conference on Communications and Electronics (ICCE). IEEE, Ha Long Bay, Vietnam, pp 52–57. <https://doi.org/10.1109/icce.2016.7562612>
22. de Bayser M, Cerqueira R (2017) Integrating MPI with Docker for HPC. In: IEEE International Conference on Cloud Engineering (IC2E). IEEE, Vancouver, BC, Canada, pp 259–265. <https://doi.org/10.1109/ic2e.2017.40>
23. Freyermuth O, Wienemann P, Bechtle P, Desch K (2021) Operating an HPC/HTC cluster with fully containerized jobs using HTCondor, Singularity, CephFS and CVMFS. *Comput Softw Big Sci* 5(1):1–20. <https://doi.org/10.1007/s41781-020-00050-y>
24. Gantikow H, Walter S, Reich C (2020) Rootless containers with Podman for HPC. In: International Conference on High Performance Computing. LNCS—ISC-HPC. Springer, Cham, pp 343–354. https://doi.org/10.1007/978-3-030-59851-8_23
25. Gerhardt L, Bhimji W, Canon S, Fasel M, Jacobsen D, Mustafa M, Porter J, Tsulaia V (2017) Shifter: containers for HPC. *J Phys Conf Ser* 898:082021. <https://doi.org/10.1088/1742-6596/898/8/082021>
26. Gomes J, Bagnaschi E, Campos I, David M, Alves L, Martins JA, Pina JA, López-García A, Orviz P (2018) Enabling rootless Linux containers in multi-user environments. The udocker tool. *Comput Phys Commun* 232:84–97. <https://doi.org/10.1016/j.cpc.2018.05.021>
27. Grupp A, Kozlov V, Campos I, David M, Gomes J, López García Á (2019) Benchmarking deep learning infrastructures by means of TensorFlow and containers. In: International Conference on High Performance Computing. LNCS—ISC-HPC, pp. 478–489. Springer, Cham. https://doi.org/10.1007/978-3-030-34356-9_36
28. Hale JS, Li L, Richardson CN, Wells GN (2017) Containers for portable, productive, and performant scientific computing. *Comput Sci Eng* 19(6):40–50. <https://doi.org/10.1109/mcse.2017.2421459>
29. Heinonen N (2019) ALCF research benefits from singularity. <https://www.hpcwire.com/off-the-wire/alcf-research-benefits-from-singularity/>. Accessed 13 Feb 2022
30. Herbein S, Dusia A, Landwehr A, McDaniel S, Monsalve J, Yang Y, Seelam SR, Taufer M (2016) Resource management for running HPC applications in container clouds. In: International Conference on High Performance Computing. LNCS—ISC-HPC, vol. 9697, pp. 261–278. Springer, Frankfurt, Germany. https://doi.org/10.1007/978-3-319-41321-1_14
31. Higgins J, Holmes V, Venters C (2016) Autonomous discovery and management in virtual container clusters. *Comput J* 60(2):240–252. <https://doi.org/10.1093/comjnl/bxw102>
32. Higgins J, Holmes V, Venters C (2016) Securing user defined containers for scientific computing. In: International Conference on High Performance Computing and Simulation (HPCS). IEEE, Innsbruck, Austria, pp 449–453. <https://doi.org/10.1109/hpcsim.2016.7568369>
33. Hisle MS, Meier MS, Toth DM (2018) Accelerating AutoDock vina with containerization. In: Proceedings of the Practice and Experience on Advanced Research Computing. ACM, New York, NY, USA. <https://doi.org/10.1145/3219104.3219154>
34. Höb M, Kranzlmüller D (2020) Enabling EASEY deployment of containerized applications for future HPC systems. In: 20th International Conference on Computational Science (ICCS). Springer, Cham, pp 206–219. https://doi.org/10.1007/978-3-030-50371-0_15

35. Hu G, Zhang Y, Chen W (2019) Exploring the performance of singularity for high performance computing scenarios. In: IEEE 5th International Conference on Data Science and Systems (DSS). IEEE, Zhangjiajie, China, pp 2587–2593. <https://doi.org/10.1109/hpcc/smartycity/dss.2019.00362>
36. Huang L, Wang Y, Lu C-Y, Liu S (2021) Best practice of IO workload management in containerized environments on supercomputers. In: Proceedings of Practice and Experience in Advanced Research Computing (PEARC). ACM, New York, NY, USA. <https://doi.org/10.1145/3437359.3465561>
37. Hursey J (2020) Design considerations for building and running containerized MPI applications. In: IEEE/ACM International Workshop on Containers and New Orchestration Paradigms for Isolated Environments in HPC (CANOPIE-HPC). IEEE, Atlanta, GA, USA, pp 35–44. <https://doi.org/10.1109/canopiehp51917.2020.00010>
38. Jacobsen DM, Canon RS (2015) Contain this: unleashing Docker for HPC. https://cug.org/proceedings/cug2015_proceedings/includes/files/pap157-file2.pdf. Accessed 13 Sept 2022
39. Jung K, Cho Y-K, Tak Y-J (2021) Containers and orchestration of numerical ocean model for computational reproducibility and portability in public and private clouds: application of ROMS 3.6. Simul Model Pract Theory 109:102305. <https://doi.org/10.1016/j.simpat.2021.102305>
40. Kadri S, Sboner A, Sigaras A, Roy S (2022) Containers in bioinformatics: applications, practical considerations, and best practices in molecular pathology. J Molec Diag 24(5):442–454. <https://doi.org/10.1016/j.jmoldx.2022.01.006>
41. Khan M, Becker T, Kuppuudaiyar P, Elster AC (2018) Container-based virtualization for heterogeneous HPC clouds: insights from the EU h2020 CloudLightning project. In: 2018 IEEE International Conference on Cloud Engineering (IC2E). IEEE, Orlando, FL, USA, pp 392–397. <https://doi.org/10.1109/ic2e.2018.00074>
42. Kovacs A (2017) Comparison of different Linux containers. In: 40th International Conference on Telecommunications and Signal Processing (TSP). IEEE, Barcelona, Spain, pp 47–51. <https://doi.org/10.1109/tsp.2017.8075934>
43. Kuity A, Peddoju SK (2017) Performance evaluation of container-based high performance computing ecosystem using OpenPOWER. In: International Conference on High Performance Computing. LNCS—ISC-HPC. Springer, Cham, pp 290–308. https://doi.org/10.1007/978-3-319-67630-2_22
44. Kumar M, Kaur G (2022) Containerized MPI application on Infiniband based HPC: an empirical study. In: 3rd International Conference for Emerging Technology (INCET), pp 1–6. <https://doi.org/10.1109/INCET54531.2022.9824366>
45. Kumar Abhishek M (2020) High performance computing using containers in cloud. Int J Adv Trends Comput Sci Eng 9(4):5686–5690. <https://doi.org/10.30534/ijatcse/2020/220942020>
46. Kurtzer GM, Sochat V, Bauer MW (2017) Singularity: scientific containers for mobility of compute. PLoS One 12(5):1–20. <https://doi.org/10.1371/journal.pone.0177459>
47. Lahiff A, de Witt S, Caballer M, La Rocca G, Pamela S, Coster D (2020) Running HTC and HPC applications opportunistically across private, academic and public clouds. EPJ Web Conf 245:07032. <https://doi.org/10.1051/epjconf/202024507032>
48. Le E, Paz D (2017) Performance analysis of applications using singularity container on SDSC Comet. In: Proceedings of the Practice and Experience in Advanced Research Computing 2017 on Sustainability, Success and Impact (PEARC). ACM, New York, NY, USA. <https://doi.org/10.1145/3093338.3106737>
49. Lee M, Ahn H, Hong C-H, Nikolopoulos DS (2022) gShare: a centralized GPU memory management framework to enable GPU memory sharing for containers. Future Gener Comput Sys 130:181–192. <https://doi.org/10.1016/j.future.2021.12.016>
50. Lim SB, Woo J, Li G (2020) Performance analysis of container-based networking solutions for high-performance computing cloud. Int J Electr Comput Eng 10(2):1507. <https://doi.org/10.11591/ijece.v10i2.pp1507-1514>
51. Liu P, Guitart J (2020) Performance comparison of multi-container deployment schemes for HPC workloads: an empirical study. J Supercomput 77(6):6273–6312. <https://doi.org/10.1007/s11227-020-03518-1>
52. Liu P, Guitart J (2022) Performance characterization of containerization for HPC workloads on Infiniband clusters: an empirical study. Clust Comput 25(2):847–868. <https://doi.org/10.1007/s10586-021-03460-8>
53. Ma H, Wang L, Tak BC, Wang L, Tang C (2016) Auto-tuning performance of MPI parallel programs using resource management in container-based virtual cloud. In: IEEE 9th International Conference on Cloud Computing (CLOUD). IEEE, San Francisco, CA, USA, pp 545–552. <https://doi.org/10.1109/cloud.2016.0078>

54. Maliszewski AM, Vogel A, Griebler D, Roloff E, Fernandes LG, Navaux Philippe OA (2019) Minimizing communication overheads in container-based clouds for HPC applications. In: IEEE Symposium on Computers and Communications (ISCC). IEEE, Barcelona, Spain, pp 1–6. <https://doi.org/10.1109/iscc47284.2019.8969716>
55. Maliszewski AM, Roloff E, Griebler D, Gasparly LP, Navaux POA (2020) Performance impact of IEEE 802.3ad in container-based clouds for HPC applications. In: International Conference on Computational Science and Its Applications (ICCSA). Springer, Cham, pp 158–167. https://doi.org/10.1007/978-3-030-58817-5_13
56. Manalo K, Baber L, Bradley R, You Z-Q, Zhang N (2019) Community collections. In: Proceedings of Practice and Experience in Advanced Research Computing (PEARC) Rise of the Machines (Learning). ACM, New York, NY, USA. <https://doi.org/10.1145/3332186.3332199>
57. Martinasso M, Gila M, Sawyer W, Sarmiento R, Peretti-Pezzi G, Karakasis V (2019) Cray programming environments within containers on cray XC systems. *Concurr Comput* 32(20):5543. <https://doi.org/10.1002/cpe.5543.e5543cpe.5543>
58. Medrano-Jaimes F, Lozano-Rizk JE, Castañeda-Avila S, Rivera-Rodriguez R (2019) Use of containers for high-performance computing. In: International Conference on Supercomputing in Mexico. CCIS—ISUM. Springer, Cham, pp 24–32. https://doi.org/10.1007/978-3-030-10448-1_3
59. Merkel D (2014) Docker: lightweight Linux containers for consistent development and deployment. <https://www.linuxjournal.com/content/docker-lightweight-linux-containers-consistent-development-and-deployment>. Accessed 13 Sept 2022
60. Michel M, Burnett N (2019) Enabling GPU-enhanced computer vision and machine learning research using containers. In: International Conference on High Performance Computing. LNCS—ISC-HPC. Springer, Cham, pp 80–87. https://doi.org/10.1007/978-3-030-34356-9_8
61. Muhtaroglu N, Ari I, Kolcu B (2018) Democratization of HPC cloud services with automated parallel solvers and application containers. *Concurr Comput* 30(21):4782. <https://doi.org/10.1002/cpe.4782>
62. Muscianisi G, Fiameni G, Azab A (2019) Singularity GPU containers execution on HPC cluster. In: Weiland M, Juckeland G, Alam S, Jagode H (eds) International conference on high performance computing. LNCS—ISC-HPC. Springer, Cham, pp 61–68. https://doi.org/10.1007/978-3-030-34356-9_6
63. Newlin M, Smathers K, DeYoung ME (2019) ARC containers for AI workloads. In: Proceedings of Practice and Experience in Advanced Research Computing (PEARC) Rise of the Machines (Learning). ACM, New York, NY, USA. <https://doi.org/10.1145/3332186.3333048>
64. Nguyen N, Bein D (2017) Distributed MPI cluster with docker swarm mode. In: IEEE 7th Annual Computing and Communication Workshop and Conference (CCWC). IEEE, Las Vegas, NV, USA, pp 1–7. <https://doi.org/10.1109/ccwc.2017.7868429>
65. Okuno S, Hirai A, Fukumoto N (2022) Performance analysis of multi-containerized MD simulations for low-level resource allocation. In: IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), pp 1014–1017. <https://doi.org/10.1109/IPDPSW55747.2022.00162>
66. Pahl C, Brogi A, Soldani J, Jamshidi P (2019) Cloud container technologies: a state-of-the-art review. *IEEE Trans Cloud Comput* 7(3):677–692. <https://doi.org/10.1109/tcc.2017.2702586>
67. Peiro Conde K (2020) Containers in HPC: is it worth it? Master's thesis. <http://hdl.handle.net/2117/335295>. Accessed 13 Sept 2022
68. Piras ME, Pireddu L, Moro M, Zanetti G (2019) Container orchestration on HPC clusters. In: International Conference on High Performance Computing. LNCS—ISC-HPC. Springer, Cham, pp 25–35. https://doi.org/10.1007/978-3-030-34356-9_3
69. Priedhorsky R, Randles T (2017) Charliecloud. In: Proceedings of International Conference for High Performance Computing, Networking, Storage and Analysis, SC '17. ACM, New York, NY, USA. <https://doi.org/10.1145/3126908.3126925>
70. Priedhorsky R, Canon RS, Randles T, Younge AJ (2021) Minimizing privilege for building HPC containers. In: Proceedings of International Conference for High Performance Computing, Networking, Storage and Analysis, SC, pp 1–14 <https://doi.org/10.1145/3458817.3476187>
71. Ramon-Cortes C, Servén A, Ejarque J, Lezzi D, Badia RM (2018) Transparent orchestration of task-based parallel applications in containers platforms. *J Grid Comput* 16(1):137–160. <https://doi.org/10.1007/s10723-017-9425-z>
72. Rudy O, Garcia-Gasulla M, Mantovani F, Santiago A, Sirvent R, Vazquez M (2019) Containers in HPC: a scalability and portability study in production biological simulations. In: IEEE International


- Parallel and Distributed Processing Symposium (IPDPS). IEEE, Rio de Janeiro, RJ, Brazil, pp. 567–577. <https://doi.org/10.1109/ipdps.2019.00066>
73. Ruhela A, Vaughn M, Harrell SL, Zynda GJ, Fonner J, Evans RT, Minyard T (2020) Containerization on petascale HPC clusters. In: Supercomp State Pract Arch. SC—SOTP. https://sc20.supercomputing.org/proceedings/sotp/sotp_files/sotp120s2-file1.pdf
 74. Ruhela A, Harrell SL, Evans RT, Zynda GJ, Fonner J, Vaughn M, Minyard T, Cazes J (2021) Characterizing containerized HPC applications performance at petascale on CPU and GPU architectures. In: International Conference on High Performance Computing. LNCS—ISC-HPC. Springer, Cham, pp 411–430. https://doi.org/10.1007/978-3-030-78713-4_22
 75. Saha P, Beltre A, Uminski P, Govindaraju M (2018) Evaluation of docker containers for scientific workloads in the Cloud. In: Proceedings of the Practice and Experience on Advanced Research Computing. ACM, New York, NY, USA. <https://doi.org/10.1145/3219104.3229280>
 76. Sampedro Z, Holt A, Hauser T (2018) Continuous integration and delivery for HPC. In: Proceedings of the Practice and Experience on Advanced Research Computing (PEARC). ACM, New York, NY, USA. <https://doi.org/10.1145/3219104.3219147>
 77. Sande Veiga V, Simon M, Azab A, Fernandez C, Muscianisi G, Fiameni G, Marocchi S (2019) Evaluation and benchmarking of Singularity MPI containers on EU research e-infrastructure. In: IEEE/ACM International Workshop on Containers and New Orchestration Paradigms for Isolated Environments in HPC (CANOPIE-HPC). IEEE, Denver, CO, USA, pp 1–10. <https://doi.org/10.1109/canopie-hpc49598.2019.00006>
 78. Simchev T, Atanassov E (2020) Performance effects of running container-based Open-MPI cluster in public cloud. In: International Conference on Large-Scale Scientific Computing (LSSC). Springer, Cham, pp 254–262. https://doi.org/10.1007/978-3-030-41032-2_29
 79. Sochat VV, Prybol CJ, Kurtzer GM (2017) Enhancing reproducibility in scientific computing: metrics and registry for Singularity containers. PLoS One 12(11):0188511. <https://doi.org/10.1371/journal.pone.0188511>
 80. Sparks J (2017) HPC containers in use. In: Proceedings of the Cray User Group (CUG). https://cug.org/proceedings/cug2017_proceedings/includes/files/pap164s2-file1.pdf. Accessed 13 Sept 2022
 81. Sparks J (2018) Enabling docker for HPC. *Concurr Comput* 31(16):5018. <https://doi.org/10.1002/cpe.5018>
 82. Steffemel LA, Charão AS, Alves B, de Araujo LR, da Silva LF (2020) MPI to Go: container clusters for MPI applications. In: International Conference on Cloud Computing and Services Science. Springer, Cham, pp 199–222. https://doi.org/10.1007/978-3-030-49432-2_10
 83. Tippit J, Hodson DD, Grimaila MR (2021) Julia and singularity for high performance computing. In: Advances in Parallel and Distributed Processing, and Applications, pp 3–15. Springer, Cham. https://doi.org/10.1007/978-3-030-69984-0_1
 84. Torrez A, Randles T, Priedhorsky R (2019) HPC container runtimes have minimal or no performance impact. In: IEEE/ACM International Workshop on Containers and New Orchestration Paradigms for Isolated Environments in HPC (CANOPIE-HPC). IEEE, Denver, CO, USA, pp 37–42. <https://doi.org/10.1109/canopie-hpc49598.2019.00010>
 85. Tronge J, Chen J, Grubel P, Randles T, Davis R, Wofford Q, Anaya S, Guan Q (2021) BeeSwarm: enabling parallel scaling performance measurement in continuous integration for HPC applications. In: 36th IEEE/ACM International Conference on Automated Software Engineering (ASE), pp 1136–1140. <https://doi.org/10.1109/ASE51524.2021.9678805>
 86. Vaillancourt PZ, Coulter JE, Knepper R, Barker B (2020) Self-scaling clusters and reproducible containers to enable scientific computing. In: IEEE High Performance Extreme Computing Conference (HPEC). IEEE, Boston, MA, USA, pp 1–8. <https://doi.org/10.1109/hpec43674.2020.9286208>
 87. Vaillancourt P, Wineholt B, Barker B, Deliyannis P, Zheng J, Suresh A, Brazier A, Knepper R, Wolski R (2020) Reproducible and portable workflows for scientific computing and HPC in the cloud. In: Practice and Experience in Advanced Research Computing (PEARC). ACM, Portland, OR, USA, pp 311–320. <https://doi.org/10.1145/3311790.3396659>
 88. Vallee G, Gutierrez CEA, Clerget C (2019) On-node resource manager for containerized HPC workloads. In: IEEE/ACM International Workshop on Containers and New Orchestration Paradigms for Isolated Environments in HPC (CANOPIE-HPC). IEEE, Denver, CO, USA, pp 43–48. <https://doi.org/10.1109/canopie-hpc49598.2019.00011>
 89. Wang Y, Evans RT, Huang L (2019) Performant container support for HPC applications. In: Proceedings of the Practice and Experience in Advanced Research Computing on Rise of the Machines (learning) (PEARC). ACM, Chicago, IL, USA. <https://doi.org/10.1145/3332186.3332226>

90. Weidner O, Atkinson M, Barker A, Filgueira Vicente R (2016) Rethinking high performance computing platforms. In: ACM International Workshop on Data-Intensive Distributed Computing. ACM, Kyoto, Japan, pp 19–26. <https://doi.org/10.1145/2912152.2912155>
91. Yang X, Kasahara M (2022) LPMX: a pure rootless composable container system. BMC Bioinf 23(1):1–13. <https://doi.org/10.1186/s12859-022-04649-3>
92. Youn C, Das AK, Yang S, Kim J (2019) Developing a meta framework for key-value memory networks on HPC clusters. In: Proceedings of the Practice and Experience in Advanced Research Computing on Rise of the Machines (Learning) (PEARC). ACM, New York, NY, USA. <https://doi.org/10.1145/3332186.3332216>
93. Younge AJ, Pedretti K, Grant RE, Brightwell R (2017) A tale of two systems: Using containers to deploy HPC applications on supercomputers and clouds. In: IEEE International Conference on Cloud Computing Technology and Science (CloudCom). IEEE, Hong Kong, pp 74–81. <https://doi.org/10.1109/cloudcom.2017.40>
94. Yuan S, Brandt SR, Chen Q, Zhu L, Salatin R, Dooley R (2020) A sustainable collaboratory for coastal resilience research. Future Gener Comput Syst 111:786–792. <https://doi.org/10.1016/j.future.2019.11.002>
95. Zhang J, Lu X, Panda DK (2016) High performance MPI library for container-based HPC cloud on InfiniBand clusters. In: 45th international conference on parallel processing (ICPP). IEEE, Philadelphia, PA, USA, pp 268–277. <https://doi.org/10.1109/icpp.2016.38>
96. Zhang J, Lu X, Panda DK (2016) Performance characterization of hypervisor-and container-based virtualization for HPC on SR-IOV enabled InfiniBand clusters. In: IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW). IEEE, Chicago, IL, USA, pp 1777–1784. <https://doi.org/10.1109/ipdpsw.2016.178>
97. Zhang J, Lu X, Panda DK (2017) Is singularity-based container technology ready for running MPI applications on HPC clouds? In: Proceedings of the 10th International Conference on Utility and Cloud Computing (UCC '17). ACM, New York, NY, USA, pp 151–160. <https://doi.org/10.1145/3147213.3147231>
98. Zhou N, Georgiou Y, Pospieszny M, Zhong L, Zhou H, Niethammer C, Pejak B, Marko O, Hoppe D (2021) Container orchestration on HPC systems through Kubernetes. J Cloud Comput 10(1):1–14. <https://doi.org/10.1186/s13677-021-00231-z>

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Springer Nature or its licensor holds exclusive rights to this article under a publishing agreement with the author(s) or other rightsholder(s); author self-archiving of the accepted manuscript version of this article is solely governed by the terms of such publishing agreement and applicable law.

Authors and Affiliations

Rafael Keller Tesser^{1,2}  · Edson Borin¹

Edson Borin
borin@unicamp.br

¹ Institute of Computing, University of Campinas, Av. Albert Einstein, 1251, Campinas, SP 13083-852, Brazil

² Center for Computing in Engineering and Sciences, University of Campinas, R. Josué Castro, s/n, Campinas, SP 13083-861, Brazil