Containers for High Performance Computing (HPC)

Juan Carlos Ramos Martinez
University of Tartu
jramos@ut.ee

Abstract
Emergent HPC experiments require complex workflows that combine Machine Learning, Big Data, and High-Performance Data Analytics. These requirements bring challenges to resource management, scheduling, and environment deployment. Container technology makes it easy to create highly portable and reproducible environments. They are currently the most popular technology for application delivery. For this reason, the research community has put effort into bringing containers to HPC but many challenges still need to be addressed. This report showcases one of those challenges: the performance difference of native and container-based HPC workloads.

1 Introduction
Supercomputing services are among the most critical services provided by supercomputing centers worldwide. This designation typically refers to using aggregated computing power to solve advanced computational problems related to scientific research. Most of the programs are executed in High-Performance Computers that provide the computational resources and tools necessary in various scientific fields. Recently, HPC centers are facing increasing demand for greater software flexibility to support faster and more diverse innovations in computational scientific work. Users and administrators face different problems with traditional HPC systems related to flexibility, scalability, integration, and portability. It is not easy to reproduce experiments from artifacts like code, data, and previous researchers’ results. The common solution to these problems is to adopt Cloud Computing environments.

The reliability and fast adoption of Cloud Computing technology have enabled the development of lightweight virtualization technologies that solve the problem of custom user-supplied code. Containerization has dramatically improved the productivity and simplicity of Cloud technologies; and, together with the advanced orchestration, enabling the adoption of Big Data software by a large community. The HPC community is involved in the transformation of adopting container virtualization to benefit from some of its well-known advantages. One is to encapsulate specific software environments for each user, allowing customization, portability, and reproducibility research [1]. Also, the isolation of users from the underlying system and other users provides for security and fault protection; the agile and fine-grain resource allocation and balancing allow for efficient cluster utilization and failure recovery [2]. A common concern is that containers may introduce performance overhead [3]. For this, few comprehensive and rigorous HPC-focused benchmarks of container performance have been addressed. The main unsolved challenges of containers on HPC remain relevant across research studies: I/O throughput optimization, multilayer monitoring, performance prediction, isolation, and data security [4].

In this report, we study the performance of different container solutions on HPC workloads. Additionally, we address the challenges required to adapt containerization approaches to HPC infrastructures, mainly by analysing the performance characteristics of memory, network and CPU. Finally, the report will answer two research questions:

- RQ1: What is the state-of-the-art and advancements in HPC and container technology?
- RQ2: What are the trade-offs of executing container-based workloads on HPC environments?

The remainder of this report is organized as follows: Section 2 covers the concepts of container runtimes in the context of HPC. Section 3 covers previous work of using containers on HPC. Section 4 discusses performance parameters considered to evaluate our research questions. Finally, Section 5 concludes the report with a discussion.

2 Background

2.1 Containerization Technology
Containerization is a lightweight, low-overhead alternative to full machine virtualization. With the introduction of Docker container engine in 2013, containerization gained tremendous popularity. Since then, several containerization techniques have been developed primarily based on chroot, control groups, and Linux namespace features. Out of the available container technologies, this work focuses on Charliecloud [5], Singularity...
Figure 1: Features of Containers

2.1.1 Podman
Podman provides a comprehensive container solution similar to Docker. The main driving force for the development of Podman was the need to avoid having a root daemon used in Docker. Podman replaces the daemon-client architecture of Docker with individual processes that run the containers and uses `conmon` to provide the necessary monitoring and debugging capabilities.

2.1.2 Singularity
Singularity is a container framework explicitly tailored for HPC environments. Its goal is to provide isolation for workloads while preventing privilege escalation, offer native support for MPI, Infini-band, and GPUs. It also supports ease of portability and reusability through the distribution of container images as a single file.

2.1.3 Charliecloud
Charliecloud provides unprivileged containers for user-defined software stacks in HPC. The motivation for developing Charliecloud was to meet the increasing demand at supercomputing centers for user-defined software stacks. These demands include complex dependencies or build requirements, externally required configurations, portability, consistency, and security. Charliecloud is a lightweight, open-source software stack based on the Linux user namespace. It uses Docker to build images, uses a shell script for automation.

3 Related Work
The question of container performance for HPC has been addressed in different ways. In recent studies, Hu et al. investigated Singularity containers’ CPU, memory, and network bandwidth. Rudyy et al. explored the scalability and portability aspects of Docker and Singularity in biological systems. At the benchmarks level on a medium system scale, Torrez et al. demonstrated minimal overhead by Charliecloud, and Singularity containers. Apart from performance studies, Canon et al. reviewed the challenges and gaps in existing containerized approaches for HPC applications. Bachiega et al. in 2019 survey on recent research and challenges revealed a lack of thorough studies involving containers and their performance in the HPC environment.

In light of this research, the focus in this report is to showcase prior experiments to establish the performance of containers in the HPC environment and HPC applications with real workloads. In particular, the works of Abraham et al., Woo and Lim, and Jha et al. are presented as references to compare the performance of their experiments in three dimensions: CPU, memory, and network.

4 Performance metrics of containerization
One of the most common tasks in this context is to evaluate the performance overhead of HPC-focused container implementations. Mainly, the observation is done over the performance of four dimensions: CPU performance, Memory performance, Application performance, and Memory usage. The following is a comparison of the most recent papers addressing these four dimensions on their benchmarks.

4.1 CPU Performance
Abraham et al. use SysBench to compute prime numbers, showing that the performance was nearly identical across four container environments with different configurations. They also run the HPCG benchmark on power-of-two node counts for the four environments showing performance tests on production systems. Their first try at this experiment yielded containers that performed almost identically, compared to the bare metal execution that was about 1.8% slower.
Target (OST)s and a single Metadata Target (MDT) \cite{13}. The workload was Sysbench to benchmark memory performance, CPU performance, and sequential and random file I/O throughput. All Sysbench benchmarks ran with eight threads, and measurements were averaged after ten runs. Their results show that for the CPU, memory, and I/O workloads across all four container solutions, they provide a rather similar performance of CPU and memory for the studied workloads.

To evaluate the CPU performance \cite{14} implemented Linpack microservices using HPC-equipped containers. Figure 4 shows the arithmetic mean with maximum and minimum values for the performance of Linpack in different scenarios. Their results show that the performance of Linpack is highest in the case of 23.81 GFLOPS, which is 24\% higher than the baseline performance. The second-highest performance they provide is for two configurations of containers with a performance gain of 19\% and 8\%, respectively. The performance gain is met because of the availability of extra computational resources not used by other microservices, thus increasing the performance of Linpack.

### 4.2 Memory Performance

STREAM memory tests were executed by Abraham et al. to show that the performance was closely identical across four environments \cite{12}. Percentages of results are normalized to median bare-metal performance at each node count. These four environments have similar performance, and the spread is higher than two bare-metal experiments, up to roughly 3\%. They made a single run of each mentioned container runtime at 1024 nodes (36,864 nodes in total). The results were: bare metal 6.73 TFLOPS, Charliecloud 6.91, Singularity 7.03, a spread of 4.5\%. These results are consistent with a hypothesis of no meaningful difference between the environments.

Jha et al. evaluate the memory performance also using the STREAM benchmark \cite{14}. They provide statistics for four-vector operations (COPY, SCALE, ADD, and TRAID) presented in Figure 5. In the COPY operation, degradation of 14\%, 15\%, and 16\% is observed for the execution of two STREAM microservices in three different configurations. This is because of the interference caused by memory-intensive operations executing together. The next worst-case performance they observe is for the case where containers share the available memory with degradation of 3\%. For other combinations, a slight performance gain is noticed with a maximum 4\% increase followed by 3\% due to the nature of their dependencies on memory. The results also show a very slight deviation from the mean value as the median is almost the same as the mean. The maximum and minimum values are also similar to the mean, except for the case of the collocated execution of two STREAM instances. Their experiment shows that the overall implementation of STREAM mi-

---

**Figure 3: STREAM Ratio values for interference of microservices in different scenarios does not significantly vary from the baseline performance. A small performance gain is achieved when STREAM is collocated with different microservices inside a container.**

---

**Figure 4: Linpack performance results**

---

On the other hand, Woo and Lim measured memory usage in three different ways using the STREAM benchmark \cite{15}. Figure 2 shows the total node memory and container implementation overhead, showing memory usage for STREAM where lower is better (less memory used). To exclude startup and teardown effects, they removed a small number of memory samples less than 40 GiB. These results show that Charliecloud adds 1200 MiB of overhead over bare metal, while Singularity adds a minimum of 37 MiB. Charliecloud’s memory cost is most likely due to its complete 1.2 GiB image residing in memory. This cost could be significantly reduced or eliminated by a new Charliecloud image. The authors conclude that the container implementation itself uses this additional memory, and implementors should verify the paper’s results and explore any efficiency gains available.
4.3 Network Performance

The Netperf microservice is used by Abraham et al. to analyze the system network performance [12]. Netperf uses client/server architecture for data transfer. During the test case, one container acts as a server that runs the net server application of netperf, while another container acts as a client running the netperf application. The experiment consists of a network performance analysis of data stream transferred from client to server for a defined duration of 120 seconds using TCP. The throughput of request-response is analyzed across defined configurations. Their results show the performance of TCP Stream presented in Figure 5. The results show that the average throughput for Netperf TCP is always affected by the execution of other microservices. On average, the maximum degradation is present for multiple microservices executing inside a container with an average performance loss of 42.8%. They also detect the worst performance with degradation of 60.4%. For other experimental cases, there is a significant performance degradation for TCP execution with an average loss of 38.3%. This is due to the case where both CPU and memory together are stressed. At the same time, TCP accesses memory and CPU resources for transferring continuous data streams, thus leading to constant performance interference.

For Jha et al., the network performance measurements in container systems employed a network benchmark IPerf tool [14]. They also implemented a set of microbenchmarks mentioned in previous studies, such as Linpak, Stream, and Netperf. They evaluated the network performance throughput in the field of Internet of Things devices, considering network performance based on container virtualization techniques. The results show that the usage of containers is the most appropriate solution and feasible in terms of response time, CPU utilization, and memory usage in this system. This is because they yield low overhead, allowing rapid deployment at a large scale. The authors also presented a solution to handle computational resources using network interface from devices involved in Fog and Edge environments, and they conclude that this variability is almost non-existent and does not affect the communication between devices. However, to enhance the performance degradation and scalability bottleneck of I/O virtualization, they also designed a middleware to reduce the bandwidth of the virtual network and collective data streams. Their results improve the performance of executing an actual scientific application.

5 Discussion and Conclusion

Both containers and virtual machines technologies enable users to define and build their software environments and then run them on top of various resources in a portable, reproducible way. This report presents an investigation of containerization technologies that are commonly explored in HPC environments. We have identified the main features of container technology. Then, we examined existing approaches and reviewed some of the current research results. We described containers architecture for computing systems by illustrating and discussing some details for application domains.

This work has shown that it is important to understand the capabilities and techniques available for a given containers-based solution and the characteristics of workloads to optimize systems. At this moment, the container approach is at the heart of the modern computing infrastructure. It avoids several challenges related to intricate execution environment dependencies that are often in conflict with other components of the application workflows. We exposed the mechanisms that use HPC containers for standard benchmarks to understand the trade-offs of HPC workloads. Furthermore, this report addressed the main features adapted to targeted HPC resources such as GPUs, Infini-band, etc. These can be used alongside containers as a base to build environments for Data Analytics on HPC. Finally, we can say that our two research questions have been answered. For RQ1, we saw that current advancements of containers for HPC are in constant development and associated container runtimes tackle different challenges of the HPC architectural model. In the same way, the RQ2 is discussed throughout the performance analysis section, showing that the trade-offs are based on three dimensions of execution performance.

As a final remark, current HPC container runtime implementations must make a difficult trade-off of the presented features for allowing nearly-native performance. The hope is that all these methods can be further inves-
tigated to find a tractable solution to performant reproducibility eventually. Developers, testers, and end-users can leverage containerization on HPC systems in a performant way, at a large scale, to reduce software development and maintenance efforts except for specific use cases involving proprietary libraries or non-compatible architectures and binary formats. The cost of performance at scale is to build support for non-portable libraries into the containers. As software complexity continues to rise along with the dependency on national supercomputing capabilities, the need for crucial container features becomes ever more critical in the search for Big Data and beyond.

References


