Assessing the Computation and Communication Overhead of Linux Containers for HPC Applications

Guilherme Rezende Alles, Alexandre Carissimi, Lucas Mello Schnorr

Instituto de Informática – Universidade Federal do Rio Grande do Sul (UFRGS)
Caixa Postal 15.064 – 91.501-970 – Porto Alegre – RS – Brazil

{gralles,asc,schnorr}@inf.ufrgs.br

Abstract. *Virtualization technology provides features that are desirable for high-performance computing (HPC), such as enhanced reproducibility for scientific experiments and flexible execution environment customization. This paper explores the performance implications of applying Operating System (OS) containers in HPC applications. Docker and Singularity are compared to a native baseline with no virtualization, using synthetic workloads and an earthquake simulator called Ondes3D as benchmarks. Our evaluation using up to 256 cores indicate that (a) Singularity containers have minor performance overhead, (b) Docker containers do suffer from increased network latency, and (c) performance gains are attainable with an optimized container on top of a regular OS.*

1. Introduction

One fundamental aspect of scientific computing research is that, in order for it to be credible, it should be reproducible. This means that the researcher has the responsibility of providing its readers with the resources necessary to run experiments and obtain similar results to the ones provided by the research. Although the concept of reproducibility is well known in the scientific community, it is very hard to put it to practice because of the large number of variables that comprise an execution environment. Additionally, HPC resources are usually managed under strict usage policies, many of which do not allow the user to customize the environment with an arbitrary software stack (e.g. compilers, libraries or device drivers), thus making it very difficult to obtain a uniform execution platform for a given experiment across different clusters.

Although being intuitively difficult to deal with, these problems can be solved through the use of traditional hardware virtualization, in which users can run their own operating system and execution environment on top of a hypervisor, ensuring that the software stack can be configured to the exact requirements for a given experiment. The problem with this approach is that, when considering HPC applications, the performance overhead introduced by the hypervisor and an entire virtual machine is too high for this solution to be considered feasible. As an alternative to virtual machines, container technologies have gained a lot of attention in the software development industry. This is mainly because containers offer the benefits of virtualization at a fraction of the performance overhead introduced by virtual machines [Chung et al. 2016]. By using containers to isolate and package software, users are able to create reproducible, configurable environments for their applications. This diminishes the hassle of managing application dependencies and software stack differences when executing experiments across different

Linux environments. The problem is that the performance implications of applying Operating System (OS) containers in parallel applications remains relatively unexplored in the HPC community.

This paper compares and contrast two container technologies, Docker and Singularity, with respect to their performance overhead and ability to solve common HPC workflow problems. We will consider the same experiments running on a native environment as the baseline for performance comparison using multiple nodes (up to 64 hosts, 256 cores). The main contributions of this paper are:

- We employ three cases: the NAS-EP [Bailey et al. 1991], the earthquake simulator Ondes3D [Dupros et al. 2010] and a in-house MPI-based Ping-Pong application to evaluate network latency (Section 4.1);
- We demonstrate that Singularity containers have minor computation overhead because of extra work during initialization; and that this overhead is absorbed for longer runs independent of the number of MPI ranks (Section 4.2);
- Docker containers with network virtualization (using the Docker Swarm) suffer from increased network latency that strongly penalizes communication-bound applications such as Ondes3D (Section 4.3);
- Performance gains are attainable with an optimized container on top of a regular OS such as our comparison of a Singularity container based on Alpine Linux that reduces the Ondes3D execution time of up to $\approx$5% in parallel runs (Section 4.4).

Section 2 presents some basic concepts regarding container technologies, Docker, and Singularity. Section 3 discusses related work in the field of containers and virtualization in high-performance computing contexts, and motivate our work. Section 4 presents the experimental design and benchmarks, and a detailed description of the main contributions of this paper. Section 5 concludes the paper with main achievements and detail some future work. The companion material of this work, including the source code, analysis scripts, and raw measurements, is publicly available at `https://github.com/guilhermealles/hpc-containers/`.

2. Basic Concepts

Operating System (OS) Containers [Soltesz et al. 2007] are a mean of achieving virtualization without relying on software to emulate hardware resources. Therefore, containers are known as software level virtualization for Linux systems, and they orchestrate features (*cgroups* and *namespaces*) that are native to the Linux kernel to isolate the resources managed by the OS. As of result, software that runs inside of a container can have its own file system, process tree, user space and network stack, giving it the impression of being executed on an isolated environment. Containers present a theoretically negligible overhead penalty when compared to an application running natively on the host operating system. This happens because the Linux kernel already uses *cgroups* and *namespaces* to manage its resources internally, even when there are not multiple containers on a single machine. Considering this approach, a non-virtualized Linux environment can be seen itself as a single container running on top of the Linux kernel, which means that there is no additional software layer in a container that should insert execution overhead. The core APIs and functionality used to create containers are not new, and have been present in the Linux kernel for more than a decade. Containers become mainstream after a long

time especially because of how difficult it is for an end user to interact with these kernel APIs directly. Conversely, containers only became popular when software (such as LXC, OpenVZ, Systemd-nspawn, Docker, rocket container runtime – rkt, and Singularity) was created to interact with the kernel and mediate the creation of containers. These container management platforms also introduced new features which are very desirable for many workflows (including software development and HPC), such as the ability to encapsulate an entire environment in an image that can be distributed and reproduced on top of different hardware, improving reproducibility and dependency management. Among all alternatives, we describe below two of them: Docker and Singularity since they are the more prominent and widely used in the OS and HPC communities.

Docker [Merkel 2014] is a very popular container system for software development and service deployment. Every major cloud infrastructure provider (such as AWS, Google Cloud Platform, and Microsoft Azure) supports Docker as a platform for executing software, and companies all over the world rely on it to deploy its services. Docker implements a virtualization model that, by default, isolates as many aspects of the underlying operating system as possible. As a result, a Docker container has many aspects that resemble a traditional virtual machine: it has its own network stack, user space, and file system. By virtualizing the network stack, Docker relies on a virtual controller that uses Network Address Translation (NAT) to correlate multiple containers to the host's IP address. This approach forces the user to explicitly specify which ports of the container should be exposed to the host operating system, allowing the user to have a finer control over network communication on the container.

Additionally, the user space is also separated between container and host. This means that there is a new root user inside the container, which is controlled by the user who starts it. This turn customization easier, for example to install libraries and packages and make modifications to the virtualized operating system. On the other hand, it also presents a security concern on shared environments, because a user can mount the root directory from the host operating system as a volume in the container, thus granting access to all the files in the host machine. Docker mitigates this issue by requiring root privileges in the host operating system for a user to create containers. Although efficient, this limitation imposes a barrier in adopting Docker as a standard container platform for shared environments in which not every user is granted with root privileges.

Singularity [Kurtzer et al. 2017] is a container system developed for scientific research and high-performance computing applications. Contrary to Docker, Singularity does not aim to create completely isolated environments. It relies on a more open model, with the objective of providing integration with existing tools installed on the host operating system. Consequently, the only namespace that is isolated between the host and a Singularity container is the file system. Other namespaces remain untouched by default. Thus, the network stack, process tree, and user space are the same between container and host, which lead to the container being seen as a process which is executed in the host operating system. This feature is very important for two reasons. First, Singularity containers can be started and killed by any tool used to manage processes, such as *mpirun* or even SLURM. Second, because the user space is untouched, the user that executes processes inside the container is the same as the one which started the container, which means that regular users can start a container without needing root access in the host OS.

3. Related Work and Motivation

Experiments to measure and evaluate the performance of virtualized environments for HPC have already been done in the past. One particular study compared the performance of Docker containers to traditional virtual machines for single-node applications, concluding that the former has a considerably lower overhead when compared to the latter [Chung et al. 2016]. The same conclusions were drawn when considering experiments that run on multiple physical nodes [Higgins et al. 2015] and with more complex application signatures that are common in HPC, such as load imbalance and communication with other processes [Beserra et al. 2015]. Additional work has also shown that there is some additional overhead when comparing the execution time of applications on top of containers to applications in the native environment (with no virtualization) [Ruiz et al. 2015].

An investigation work proposed a model of MPI cluster in a distributed environment [Nguyen and Bein 2017]. In this study, Docker containers are connected through an orchestrator called Docker Swarm, which is responsible for assigning names and providing network connectivity between the containers, leveraging Docker's overlay networking capabilities. Performance analysis, however, is absent from this study, obscuring the conclusion of whether such an approach is viable in a real-world scenario. Furthermore, it has been shown that the performance of network operations can be affected by the use of Docker containers, especially in latency-sensitive scenarios [Felter et al. 2015].

Singularity [Kurtzer et al. 2017] is a container system designed for scientific research workflows, and it strives to solve some drawbacks of using Docker in HPC. Author argues that Docker is not designed for shared, multi-user environments (as discussed in Section 2), something very common in supercomputers. As a consequence, it is very hard to find HPC centers that allow users to execute Docker containers. Singularity, on the other hand, solves these problems to make HPC containers more accessible to the scientific community. Consequently, Singularity containers are already accepted and used in many supercomputers around the world. Additionally, a performance analysis of applications running on top of Singularity containers has also been carried out [Le and Paz 2017]. It concludes that while some overhead does exist, the reported values are negligible for most use cases.

Motivation: The goal of this work is to study the drawbacks and improvements that occur by applying virtualization techniques to high-performance computing workflows. As concluded by previous work, using virtual machines is unfeasible because of the overheads that comes along with this strategy. Thus, our goal is to measure the performance impact of applying container-based virtualization to these HPC workloads. We present an analysis covering both synthetic benchmarks and a real application comparing the performance implications of Docker and Singularity, two major container systems, and using a traditional approach (with no virtualization) as baseline. Furthermore, we intend to demonstrate that virtualization techniques can be used in HPC without the massive overhead of traditional virtual machines. Next section details our results toward these goals.

4. Results and Evaluation of the Performance Overhead

Results are based on measurements obtained from experiments with multiple compute nodes of the Grid5000 platform [Balouek et al. 2013], in a controlled setup. In what follows, we present (a) the software/hardware stack adopted across all experiments with

three cases (Native, Docker, Singularity) and three benchmarks (NAS-EP, Ondes3D, Ping-Pong); (b) the computation overhead analysis with a comparison between docker, singularity, and native; (c) a verification of the increased communication latency leading to bad application performance; and (d) a comprehensive analysis to verify how performance gains can be used solely in applying an optimized container on top of an optimized OS.

4.1. Software/Hardware Environment, Benchmarks, and Workload Details

The Grid5000 is a platform used for scientific experiments in parallel computing, HPC, and computer science. It provides its users with many clusters that can be reserved for exclusive use for a limited time. We executed the experiments in the Grid5000's `graphene` cluster (at Nancy - France), which contains 131 nodes, each one equipped with 16GB of DDR3 memory and a quad-core Intel Xeon X3340 (Lynnfield, 2.53GHz), and interconnected by a 1 Gigabit Ethernet and a 20 Gbps Infiniband network. We used up to 64 compute nodes for our tests using exclusively the 1 Gigabit Ethernet because of limitations in the container configuration. In all experiments, each node received a maximum of 4 MPI processes due to the 4-core availability of processor cores. All compute nodes are initially deployed (using `kadeploy3` [Jeanvoine et al. 2013]) with the default Debian9 OS image, before laying the Docker or Singularity environment on top of it.

To ensure consistency between the container environments against the Native case, the same Debian9 Linux distribution was used for such environments in both Docker and Singularity containers. We have used a previously proposed [Nguyen and Bein 2017] multi-node container infrastructure for Docker where physical nodes are connected using the Docker Swarm utility. This tool is responsible for spawning containers on all the nodes and connecting them via an overlay network, so that every container (which will execute an MPI process) can be addressed by the MPI middleware. The multi-node container infrastructure for Singularity is similar to the one with native processes. Because Singularity containers share the network stack with its host, there is no need for a virtual network between the containers. Therefore, processes in Singularity containers communicate through the physical network.

Three parallel applications are used to evaluate the performance in the OS options (Native, Docker and Singularity): NAS-EP, Ondes3D, and Ping-Pong, detailed as follows. The NAS Embarrassingly Parallel – **NAS-EP** – is part of the NAS Parallel Benchmarks (NPB) [Bailey et al. 1991]. NAS-EP generates independent Gaussian random numbers using the polar method, being considered a CPU-bound case with parallel speedup close to ideal since communication takes place in the beginning and end of the execution. EP is executed with the class B workload using one to four hosts (4 to 16 cores) in preliminary tests. **Ondes3D** [Dupros et al. 2010] is developed at the BRGM (French Geological Survey) as an implementation of the finite-differences method (FDM) to simulate the propagation of seismic waves in three-dimensional media. As previously observed [Tesser et al. 2017], its signature contains characteristics such as load imbalance and frequent asynchronous small-message communications among MPI ranks. Two workloads have been used to run Ondes3D: the default test case without a geological model (synthetic earthquake) and a real Mw 6.3 earthquake that arose in Liguria (north-western Italy) in 1887 [Aochi et al. 2011] with 300 timesteps, which has been used as workload for our tests. So, this real-world application is also evaluated to verify if it is impacted by OS containers. Finally, an in-house **Ping-Pong** benchmark developed with MPI (see the

companion material for the source code) was used to assess the bandwidth and latency performance when introducing the container's virtual environment. This evaluation is conducted between two nodes that exchange MPI messages, with message sizes varying from 1Byte to 1MByte.

We generate two randomized full factorial designs [Jain 1991] to drive experiments and collect measurements for NAS-EP and Ondes3D. The first design targets a smaller scale test using up to four nodes, with 1, 4, 8, and 16 processes; the second design uses 64 nodes, with 64, 128, 192, and 256 processes. The first batch uses NAS-EP executed with the Class B workload (identified by NAS-EP/ClassB) and Ondes3D with the default test case (Ondes3D/Default). The second batch of experiments considers the Ondes3D application using the Ligurian workload (Ondes3D/Ligurian). The Ping-Pong application has been used in a separated batch since it uses only two compute nodes of the graphene cluster. Messages size corresponding to powers of two from 1B to 1MBytes (21 data points) has been sequentially measured. All reported makespan and ping-pong measurements are averages from 10 to 30 replications of each experiment parameter configuration; error bars are calculated considering a confidence level of 99.7% assuming a Gaussian distribution.

4.2. Computation Overhead Analysis

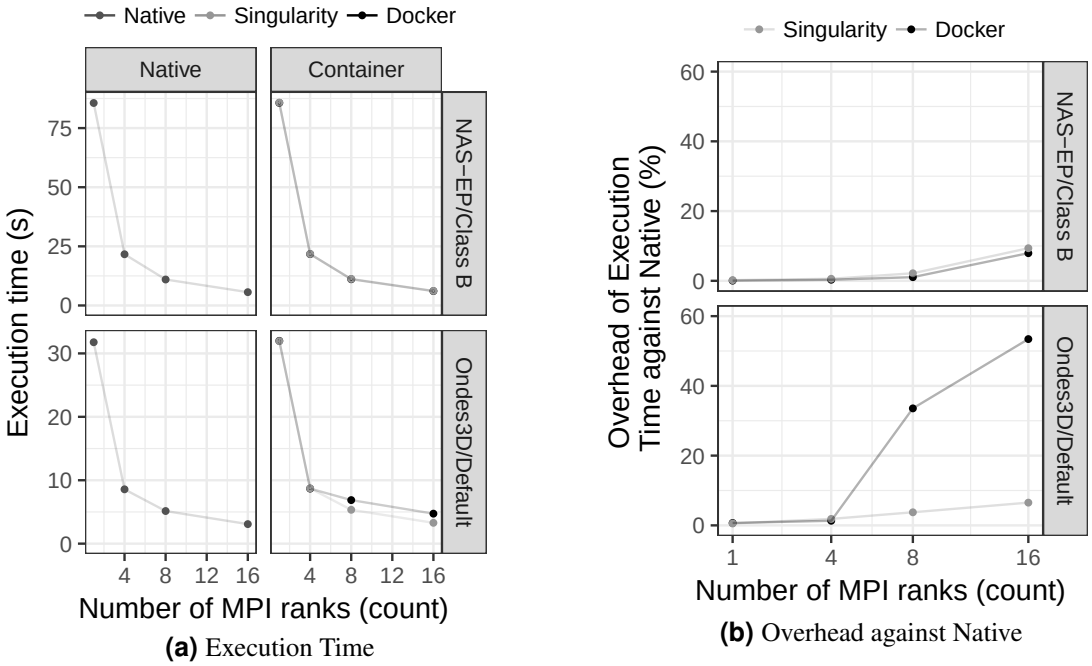
We present the results of the computation overhead for the small case scenarios (up to 16 cores) of NAS-EP/ClassB and Ondes3D/Default, then the larger scenario with the Ondes3D/Ligurian case, using 64 nodes and 256 cores.

Small case (4 nodes, 16 cores) with NAS-EP/ClassB and Ondes3D/Default

Figure 1a shows the makespans of the NAS-EP/ClassB (the top facets in the first row) and the Ondes3D/Default (bottom), with respect to the number of MPI ranks for the Native (left facets) and the Containers (right) – Singularity and Docker. Figure 1b depicts the execution time overhead with respect to the number of MPI ranks, calculated for each container environment against the native runs, also for both applications.

For the **NAS-EP/ClassB** case, Figure 1 (top facets) shows that the virtualized approaches perform very close to each other and to the native baseline. For 16 MPI ranks, the Docker overhead is of 8% while the Singularity imposes a slightly higher overhead of 9%. Although limited, both indicate an alarming increasing trend. This difference in execution time can be related to the time needed to spin up the containers and should increase as the number of containers (and MPI ranks) increases. However, since these runs were short – less than 7s for 16 processes in NAS-EP/ClassB (see Figure 1a) – such overhead may be absorbed with longer CPU-bound runs that make a limited use of the communications. For the **Ondes3D/Default** case (bottom facets of Figures 1a and 1b), we observe that the performance on the three environments is similar for 1 and 4 MPI ranks. However, the Docker performance degrades when going up to 8 and 16 ranks with execution time overhead of $\approx$33% for 8 MPI ranks and $\approx$53% for 16. This behavior surfaces exactly when more physical nodes are added to the experiment, which indicates that the network communication might be impacting the performance of Docker containers. This hypothesis is further supported by the virtual network (Docker Swarm)

that is required to provide connectivity between Docker containers. Such a virtual network does not exist in the other two environments. Although the Singularity container poses some overhead (≈6% for 16 ranks), we believe it has the same reason for the NAS-EP/ClassB case, so unrelated to the network.



(a) Execution Time

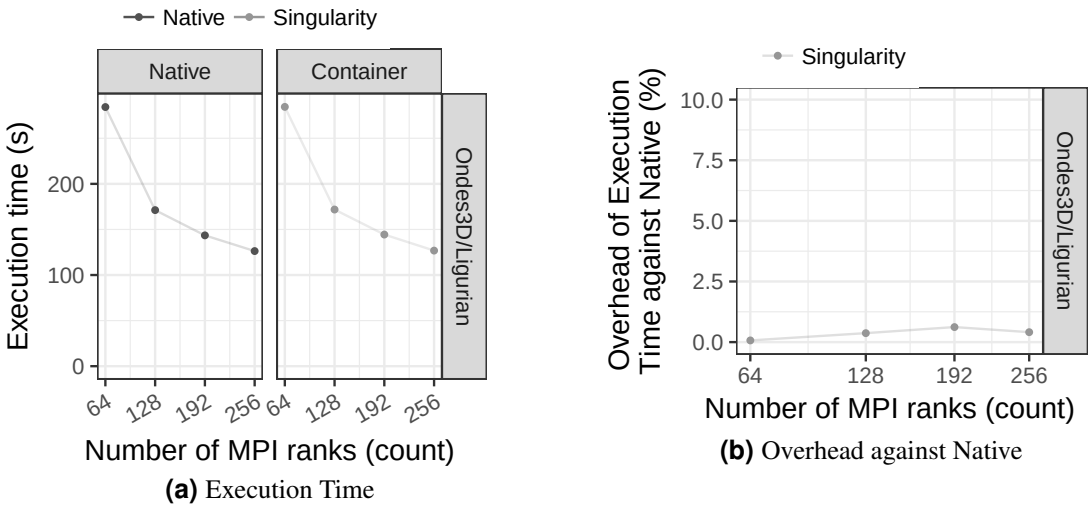
(b) Overhead against Native

Figure 1. (a) Execution time as a function of the number of MPI ranks for the three environments (Native, Docker, and Singularity) and the applications (NAS-EP/ClassB, and Ondes3D/Default), and (b), the execution overhead of the container environments against the native environment with respect to the number of MPI ranks.

Large case (64 nodes, 256 cores) with Ondes3D/Ligurian

Figure 2 shows a large-scale simulation of the Ligurian earthquake on Ondes3D. This experiment was conducted to put Singularity in a highly-distributed computing scenario, and its main objective is to assess the aggregated overhead of spawning a large number of containers across multiple nodes. Unfortunately, the container infrastructure for Docker using its overlay network and Docker Swarm as an orchestrator failed to spawn containers in such a high number of nodes, and thus Docker was excluded from this test case. As the plot indicates (see Figure 2a), there is no observable difference in execution time between the two approaches (Singularity and Native), which indicates that the additional cost of executing applications in a Singularity environment is negligible even when spawning a high number of containers. The Figure 2b shows the computed overhead as a function of the number of MPI ranks, revealing a minor overhead of less than ≈1% in all cases. We believe the overhead is minor because of the longer run (more than 100s), so any initialization time imposed by the container is more easily absorbed by the run. Surprisingly, the overhead is smaller with 256 ranks, breaking the upward trend from 64 to 192. This is probably due to the diminishing returns on speedup as the number of cores increase,

which can mask the container initialization time with other overheads that are the constant in both environments, such as network communication.



(a) Execution Time

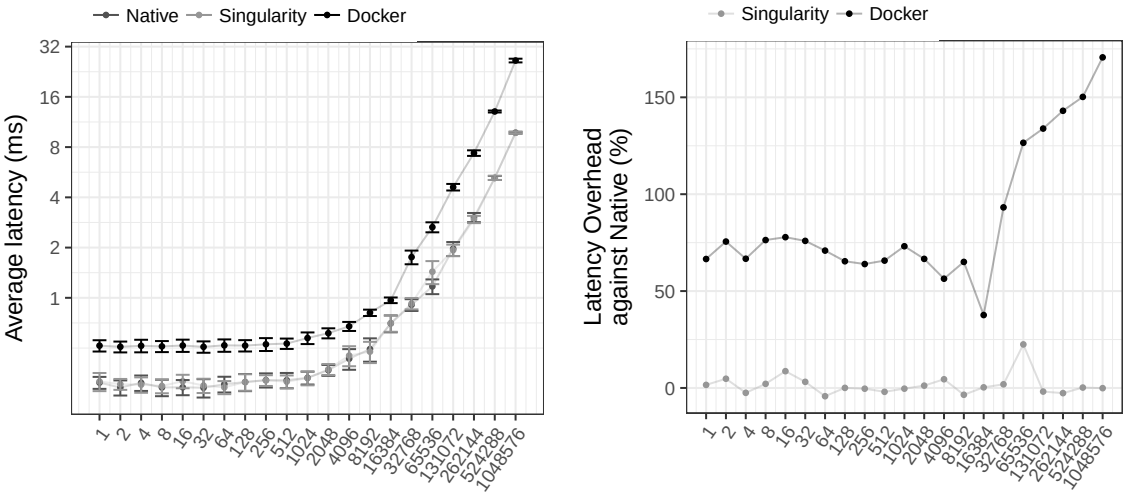
(b) Overhead against Native

Figure 2. (a) Execution time as a function of the number of MPI ranks for the three environments (Native and Singularity) with the Ondes3D/Ligurian application, and (b), the Singularity overhead with respect to the number of MPI ranks.

4.3. Verification of Increased Communication Latency

The results obtained with Ondes3D/Default using the Docker environment (see previous Subsection) led us to design an experiment to demonstrate that the bad performance is caused by network issues. Figure 3 presents the Ping Pong benchmark measure the communication latency from the application point of view. Figure 3a depicts the average latency (on the logarithmic scale Y axis) between two nodes for the three environments (differentiated by color) as a function of the message size (on X, also log scale). From these results, we can see that the Docker network latency is much higher when compared to both the native and singularity environments, therefore with poorer performance. This evidence confirms that, as observed in the Ondes3D/Default experiment, the virtual network (Docker Swarm) used by Docker introduces significant overhead to communication. Singularity containers, on the other hand, use the same network stack as the host operating system, resulting in non-observable performance differences since most of the average latency is within the confidence interval of native measurements. The Figure 3b shows the latency overhead of each container environment against the native physical interconnection. We can see that the overhead imposed by Singularity in the communication latency remains stable no matter the message size, which is something desirable. In some cases the Singularity overhead is negative, meaning that average latency measured within Singularity is smaller than the average with the native OS. This is just an artifact since we show (see Figure 3a), that confidence intervals of Singularity and Native overlap, indicating no statistical difference. The case for Docker is much worse because (a) the overhead is $\approx$75% against native, and (b) it dramatically increases after the message size 32KBytes. This indicates the low scalability of the approach, especially for those applications with larger message sizes, but also impacting applications that mostly used smaller messages. For instance, in the case of Ondes3D previously studied, ranks exchange multiple small

message according to the domain decomposition. Even if most messages are exchanged asynchronously, the latency impact on the application is easily observed (see Figure 1b).

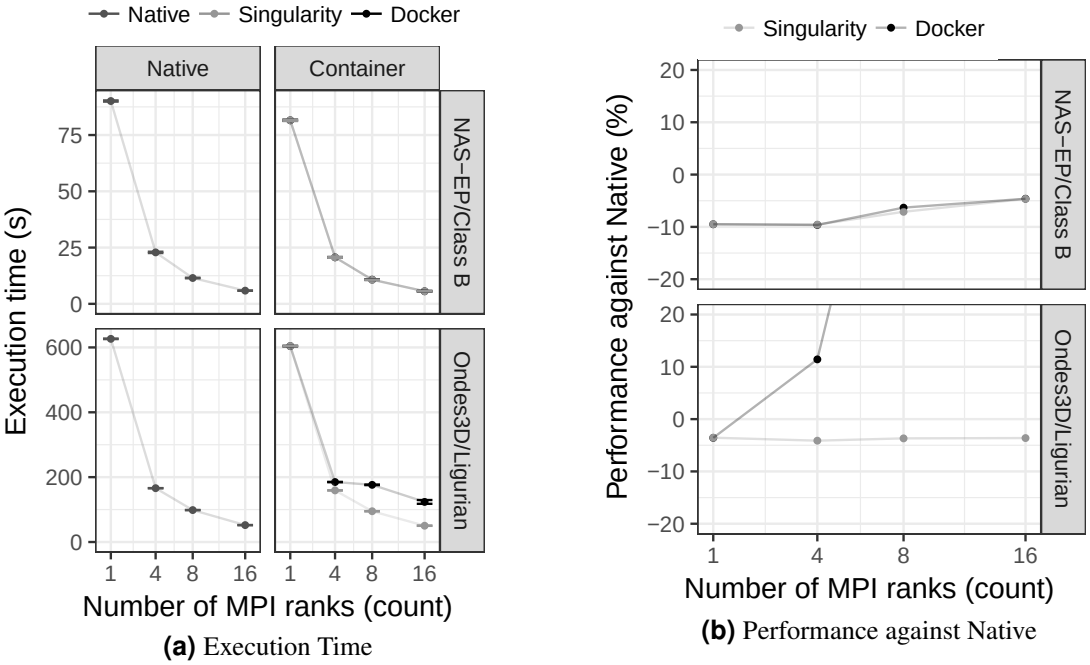


(a) Average Latency (ms) **(b)** Latency Overhead

Figure 3. (a) Average network latency (on Y log scale) measured with the Ping Pong benchmark for the three environments (color) as a function of message size (on X log scale), and (b), the same but showing the derived latency overhead against native (on Y).

4.4. Performance Gains with an Optimized Container based on Alpine Linux

To illustrate the advantages in flexibility for environment configuration, we also conducted an experiment running an Alpine Linux image on the container environments (Singularity and Docker). The Alpine Linux is a lightweight Linux distribution that strives for efficiency and isolation. It is based on Busybox [Wells 2000] and provides an alternate set of standard libraries that can yield better performance for some applications. Installing a completely different Linux distribution on multiple hosts of a cluster for a single experiment is generally a very hard task, sometimes even considered unfeasible (especially in a shared cluster environment). However, this task can be easily done when using containers. Figure 4 shows how Docker and Singularity (running the Alpine Linux distribution) compare to the native operating system (running Debian) both in terms of average execution time (Figure 4a) and performance difference against the native host (Figure 4b) as a function of the number of MPI ranks. These results show that, by modifying the execution environment, it is possible for the virtualized execution to outperform the native one. In the Ondes3D/Ligurian case, Singularity/Alpine is $\approx$5% faster than native, while in the NAS-EP/ClassB, both Docker and Singularity running Alpine are from $\approx$5% (with 16 cores) to $\approx$10% (sequential) faster than the native host when equipped with Debian9. Such results are not so surprising but are still unconventional. This experiment shows that using a fine-tuned, HPC-tailored container in experiments can bring performance advantages as well as a reproducible environment.



(a) Execution Time

(b) Performance against Native

Figure 4. (a) Execution time for the NAS-EP/ClassB benchmark and Ondes3D/Ligurian with containers running Alpine Linux while the host is running Debian9, and (b), performance difference of Alpine Linux containers (Singularity and Docker) against the Debian9 Native environment. Negative percentages in (b) indicate the performance gains of the containers.

5. Conclusion

In this paper, we assess the performance implications of the adoption of Linux Containers for HPC applications with different workloads. While containers provide similar features as hardware level virtualization with a theoretically negligible performance overhead, making them suitable for high-performance applications has to be evaluated prior using in production environments. Therefore, we compared two container technologies, Docker and Singularity, against a native environment running with no virtualization.

The results for the proposed tests indicate that containers introduce very little (if any) computation overhead in CPU-bound applications, for both Docker and Singularity. This can be verified by the lack of a clear performance difference on the EP-NAS/ClassB Benchmark among the container and native environments. Although we observed penalties up to $\approx$9%, they are completely absorbed if the applications last longer. Communication overhead, on the other hand, has been observed in Docker containers. This is mainly because the Docker architecture requires the containers to be connected through an overlay network in order for them to have connectivity across multiple hosts (which was needed for the MPI cluster). This overhead was observed in both the Ping Pong test case as well as the Ondes3D application, which is known to require frequent communication between MPI processes. Singularity is free from such overheads. Additionally, we conducted experiments that leveraged the potential flexibility that a virtualized workflow provides. Because containers allow users to fine-tune the execution environment more easily, it was possible to use a different Linux distribution without having root access to the host operating system. This approach yielded better performance than the native exe-

cution, which means that it is possible to use these fine-tuning capabilities to considerably enhance the performance of HPC applications.

With our experiments, we can conclude that Linux containers are a suitable option for running HPC applications in a virtualized environment, without the drawbacks of traditional hardware-level virtualization. In our tests, we concluded that Singularity containers are the most suitable option both in terms of system administration (for not granting every user that starts a container root access to the system) and in terms of performance (for not imposing an overlay network that is a potential bottleneck).

As future work, we plan to investigate HPC applications within containers that make use of low-latency Infiniband networks and multi-GPU systems. We also intend to verify in further details if the computation signature of HPC codes are the same outside and inside the container.

Acknowledgements

We thank these projects for supporting this investigation: FAPERGS GreenCloud (16/488-9), the FAPERGS MultiGPU (16/354-8), the CNPq 447311/2014-0, the CAPES/Brafitec EcoSud 182/15, and the CAPES/Cofecub 899/18. Experiments were carried out at the Grid'5000 platform (`https://www.grid5000.fr`), with support from Inria, CNRS, RENATER and several other french organizations. The companion material is hosted by GitHub for which we are also grateful.

References

[Aochi et al. 2011] Aochi, H., Ducellier, A., Dupros, F., Terrier, M., and Lambert, J. (2011). Investigation of historical earthquake by seismic wave propagation simulation: Source parameters of the 1887 m6. 3 ligurian, north-western italy, earthquake. In L'Association Française du Génie Parasismique (AFPS), editor, *8ème colloque AFPS, Vers une maitrise durable du risque sismique*.

[Bailey et al. 1991] Bailey, D. H., Barszcz, E., Barton, J. T., Browning, D. S., Carter, R. L., Dagum, L., Fatoohi, R. A., Frederickson, P. O., Lasinski, T. A., Schreiber, R. S., et al. (1991). The nas parallel benchmarks. *The International Journal of Supercomputing Applications*, 5(3):63–73.

[Balouek et al. 2013] Balouek, D., Carpen Amarie, A., Charrier, G., Desprez, F., Jeannot, E., Jeanvoine, E., Lèbre, A., Margery, D., Niclausse, N., Nussbaum, L., Richard, O., Pérez, C., Quesnel, F., Rohr, C., and Sarzyniec, L. (2013). Adding virtualization capabilities to the Grid'5000 testbed. In Ivanov, I. I., van Sinderen, M., Leymann, F., and Shan, T., editors, *Cloud Computing and Services Science*, volume 367 of *Communications in Computer and Information Science*, pages 3–20. Springer International Publishing.

[Beserra et al. 2015] Beserra, D., Moreno, E. D., Endo, P. T., Barreto, J., Sadok, D., and Fernandes, S. (2015). Performance analysis of lxc for hpc environments. In *2015 Ninth International Conference on Complex, Intelligent, and Software Intensive Systems*, pages 358–363.

[Chung et al. 2016] Chung, M. T., Quang-Hung, N., Nguyen, M. T., and Thoai, N. (2016). Using docker in high performance computing applications. In *2016 IEEE Sixth International Conference on Communications and Electronics (ICCE)*, pages 52–57.

[Dupros et al. 2010] Dupros, F., Martin, F. D., Foerster, E., Komatitsch, D., and Roman, J. (2010). High-performance finite-element simulations of seismic wave propagation in three-dimensional nonlinear inelastic geological media. *Par. Comput*, 36(5):308 – 325.

[Felter et al. 2015] Felter, W., Ferreira, A., Rajamony, R., and Rubio, J. (2015). An updated performance comparison of virtual machines and linux containers. In *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 171–172.

[Higgins et al. 2015] Higgins, J., Holmes, V., and Venters, C. (2015). Orchestrating docker containers in the hpc environment. In Kunkel, J. M. and Ludwig, T., editors, *High Performance Computing*, pages 506–513, Cham. Springer International Publishing.

[Jain 1991] Jain, R. (1991). *The art of computer systems performance analysis - techniques for experimental design, measurement, simulation, and modeling.* Wiley professional computing. Wiley.

[Jeanvoine et al. 2013] Jeanvoine, E., Sarzyniec, L., and Nussbaum, L. (2013). Kadeploy3: Efficient and scalable operating system provisioning for clusters. *USENIX; login:*, 38(1):38–44.

[Kurtzer et al. 2017] Kurtzer, G. M., Sochat, V., and Bauer, M. W. (2017). Singularity: Scientific containers for mobility of compute. *PLOS ONE*, 12(5):1–20.

[Le and Paz 2017] Le, E. and 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*, PEARC17, pages 66:1–66:4, New York, NY, USA. ACM.

[Merkel 2014] Merkel, D. (2014). Docker: lightweight linux containers for consistent development and deployment. *Linux Journal*, 2014(239):2.

[Nguyen and Bein 2017] Nguyen, N. and Bein, D. (2017). Distributed mpi cluster with docker swarm mode. In *2017 IEEE 7th Annual Computing and Communication Workshop and Conference (CCWC)*, pages 1–7.

[Ruiz et al. 2015] Ruiz, C., Jeanvoine, E., and Nussbaum, L. (2015). Performance evaluation of containers for hpc. In Hunold, S., Costan, A., Giménez, D., Iosup, A., Ricci, L., Gómez Requena, M. E., Scarano, V., Varbanescu, A. L., Scott, S. L., Lankes, S., Weidendorfer, J., and Alexander, M., editors, *Euro-Par 2015: Parallel Processing Workshops*, pages 813–824, Cham. Springer International Publishing.

[Soltesz et al. 2007] Soltesz, S., Pötzl, H., Fiuczynski, M. E., Bavier, A., and Peterson, L. (2007). Container-based operating system virtualization: A scalable, high-performance alternative to hypervisors. *SIGOPS Oper. Syst. Rev.*, 41(3):275–287.

[Tesser et al. 2017] Tesser, R. K., Schnorr, L. M., Legrand, A., Dupros, F., and Navaux, P. O. A. (2017). Using simulation to evaluate and tune the performance of dynamic load balancing of an over-decomposed geophysics application. In *European Conference on Parallel Processing*, pages 192–205. Springer.

[Wells 2000] Wells, N. (2000). Busybox: A swiss army knife for linux. *Linux J.*, 2000(78es).