Assessing the Computation and Communication Overhead of Operating System Containers for HPC Applications

Guilherme Rezende Alles[1], **Alexandre Carissimi**[1], **Lucas Mello Schnorr**[1]

[1] Graduate Program in Computer Science (PPGC/UFRGS), Porto Alegre, Brazil

Abstract. *This paper explores the usage of container technologies to solve two relevant problems in high performance computing (HPC) applications. The first problem is reproducibility, expressed by the difficulty of precisely reproducing an execution environment to verify, validate or build on previous work. Because of the amount of potential dependencies and external variables an application depends upon, it can be extremely difficult to simulate the same software environment to reproduce previous research. The second problem is derived from the fact that HPC resources (e.g. clusters and grids) usually have very strict usage policies, that do not allow users to install arbitrary software to run their applications.*

Resumo. *Coloque o resumo aqui.*

1. Introduction

One fundamental problem 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 audience 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 amount 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 and device drivers), thus making it very difficult to obtain an uniform execution platform across different clusters.

Although presented as difficult to deal with, these problems can be solved through the use of traditional hardware virtualization. By abstracting the hardware layer and providing a general, well known API to the software stack, users can run their own operating system and execution environment on top of a hypervisor, ensuring that the software stack will always be the same regardless of any hardware change. The problem with this approach is that, when considering HPC applications, the performance overhead introduced by the hypervisor is too high for this solution to be considered feasible.

Lately, container technologies have gained a lot of attention in the software development industry. This is mainly because containers are presented as a solution to package enterprise software in a controlled, static environment that will run on a wide variety of Linux systems. When compared to traditional virtual machines, containers have the advantage of not needing a hypervisor for virtualization, yielding performance extremely close to native but still providing application and software stack isolation.

By using containers to isolate and package HPC applications, we expect to encourage users to use containers to create reproducible, user configurable environments for their applications, thus diminishing the hassle of managing application dependencies and software stack differences when executing experiments across multiple HPC clusters.

In fact, this approach has already been used to a great extent in the software development industry. Container technologies such as Docker are used daily to deploy software microservices to cloud environments, and cloud providers such as AWS and Google Cloud Platform offer services to manage applications based on containers. In fact, Google announced in 2014 that all of its services run on top of containers, adding to more than 2 billion containers launching every day.

Considering the large adoption of containers in software development, we compare and contrast two container technologies, Docker and Singularity, with respect to their performance overhead and ability to solve common HPC workflow problems. For the performance overhead measurements, we will consider the same experiments running on a native environment as the baseline.

2. Related Work and Motivation

2.1. Related Work

As presented in [**?**], the virtualization of HPC applications is viable in a low overhead environment such as Docker. In this study, the author compares the compute performance of Docker containers to virtual machines, concluding that the former has a considerably lower overhead when compared to the latter. This study, however, only explores the performance of single node applications, and does not present information on how containers can scale to distributed environments.

For multi node computations, [**?**] proposes the creation of an MPI cluster using Docker containers in a distributed environment. In this proposal, the containers are connected through an orchestrator called Docker Swarm, which is responsible for assigning names and providing an overlay network for transparent connectivity between the containers. Performance analysis, however, is absent from this study, obscuring the conclusion of whether such an approach is viable in a real world scenario.

In [**?**], the author introduces Singularity. Singularity is a container system designed for scientific research workflows, and it strives to solve some drawbacks of using Docker containers in HPC. The author discusses how Docker was not designed for shared, multi-user environments (such as supercomputers and HPC centers) and thus presents significant security issues when used in this context. As a consequence, it is very hard to find HPC centers that allow users to use Docker containers. Singularity, on the other hand, was built with these problems in mind and solves them in order to make HPC containers accessible to the scientific community. As a consequence, Singularity containers are already accepted and used in many supercomputers around the world. Additionally, [**?**] presents some performance analysis of applications running on top of Singularity containers. It concludes that while some overhead do exist, the reported values are negligible for most use cases.

2.2. Motivation

Our objectives for 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 not a feasible approach because of the performance overhead that comes along with this strategy. Thus, our goal is to measure the performance impact of applying virtualization in the form of container technologies to these workloads. We present an analysis covering both synthetic benchmarks and a real application comparing the usability of two major container systems - Docker and Singularity - using a traditional approach (with no virtualization) as a baseline.

Furthermore, we intend to demonstrate that virtualization techniques can be used in HPC without the massive overhead of traditional virtual machines. By using containers, cluster administrators can provide flexibility, portability and enhanced reproducibility to its users without sacrificing performance and security.

3. Background and Experimental Context

3.1. Background

3.1.1. Containers

Containers are a mean of achieving virtualization without relying on software to emulate hardware resources. Instead, containers are known as software level virtualization for Linux systems, and they use features that are native to the Linux kernel (namely, *cgroups* and *namespaces*) to isolate the resources managed by the operating system. As a 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 a completely isolated environment.

By using native kernel features to grant isolation, 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 to insert execution overhead.

In spite of being receiving large amounts of attention lately, the core APIs and functionality used to create containers is not new, and have been present in the Linux kernel for more than a decade. However, the popularization of containers took a long time to happen 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 Docker and Singularity) was created to interact with the kernel and mediate the creation of containers.

These container management platfors also introduced new features which were 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 reproduced on top of different hardware, improving reproducibility and dependency management.

3.1.2. Docker

Talk about Docker's virtualization model - in a nutshell, virtualize everything that is possible. Communication between client and daemon through UNIX socket owned by the root user.

3.1.3. Singularity

Singularity's virtualization model - virtualize only the essentials (that is, the file system). Anything else must be explicitly specified.

3.2. Experimental Context and Workload Details

The experiments were executed in the Grid5000 hardware stack. The Grid5000 is a grid platform used for scientific experiments in parallel computing, HPC and computer science. It provides its users with a large amount of clusters that can be reserved for exclusive use for a limited time. For this paper, we executed the experiments in the Grid5000's *graphene* cluster, which contains 16GB of DDR3 memory and a quad core Intel Xeon X3340 on each node. We used up to {**INSERT NODES HERE**} nodes for our tests.

The nodes were loaded with a Debian 9 image using the *kadeploy3* tool. The same distribution was used for the virtualized environments in both Docker and Singularity containers. For benchmarks, we selected the following set of applications: NAS EP, Ondes3D and Ping Pong.

The NAS EP is an application included in the NAS Parallel Benchmarks which simulates a parallel random number generator. Its name stands for *embarassingly parallel*, and it was chosen to simulate a highly CPU bound scenario with parallel speedup close to ideal.

Ondes3D is a fluid dynamics simulation application with characteristics such as load imbalance and frequent communication between MPI nodes. It was chosen to simulate a real world scenario.

Finally, the Ping Pong benchmark was used to measure the network performance and overhead when introducing the container's virtual environment.

The container infrastructure for Docker was built with the cluster proposed by [**?**]: containers were connected using Docker Swarm and an overlay network.

The container infrastructure for Singularity is pretty much the same as the one with native processes. The only difference is that instead of distributing the application binary, I distributed the container image.

Talk about the experimental design: how were the results obtained and interpreted?

4. Results

Show experiment plots (approx 3 to 4 plots)

Discuss come conclusions derived from the results

5. Conclusion

Summarize what has been discussed: the original problem, why containers might be a relevant solution, which technologies were tested

By looking at the experiment results, what are the conclusions that can be taken with respect to the performance overhead introduced by containers?

All things considered, is it okay to use containers for the kinds of application we tested? If it is (yes, it is), then which technology is the best one? (Singularity).

Why is Docker not suitable for HPC environments and especially distributed MPI applications?

Acknowledgements

Who paid for this?