Case Studies of executing containerized scientific applications on High-Performance Computing Platforms using RADICAL-Cybertools

George Koubbe - g.koubbe@rutgers.edu
Faculty Evaluator: Dr. Shantenu Jha | Technical Supervisor: Dr. Matteo Turilli
Rutgers, The State University of New Jersey
Department of Electrical and Computer Engineering
Piscataway, NJ 08854 USA

***Abstract*- The focus of this work is to provide a technical introduction to the topic of Singularity containers, and how they can be used to containerize High-Performance Computing (HPC) applications. First, we will briefly describe another container platform, Docker, in order to establish the advantages that Singularity offers when using it on HPC machines. Furthermore, we will show a characterization of the execution performance using Singularity and RADICAL-Cybertools (RCT) with and without containerization on MPI/non-MPI applications. Lastly, we proceed to analyze case studies on how to use Singularity containers for projects supported via RCT.**

Keywords: Container, Singularity, Docker, HPC, Performance evaluation, Overhead

I. INTRODUCTION

A. *RADICAL-Cybertools*

In order to support science on a range of high-performance and distributed computing systems, RADICAL-Cybertools (RCT) was created. It is a set of python libraries that support the development and execution of scalable workflows of different scientific applications.

Based on a building-block approach [1], it currently consists of three components: (i) RADICAL-SAGA: provides a homogeneous programming interface to the majority of production High-Performance Computing (HPC) queuing systems, Grid- and Cloud-services, (ii) RADICAL-Pilot (RP): uses pilots (jobs submitted to a machine in order to acquire exclusive use of a chunk of its resources) to achieve the scalable execution of large numbers of tasks, (iii) RADICAL-EnTK: offers the ability to create and execute ensemble-based workflows with different stages and tasks with complex communications between every stage but without the need for explicit resource management.

B. *Container Technologies*

The idea of containers involves what is known as Operating System (OS) and application virtualization [2], and it became popular since 2013 with the growing interest of cloud providers and Internet Service Providers (ISP).

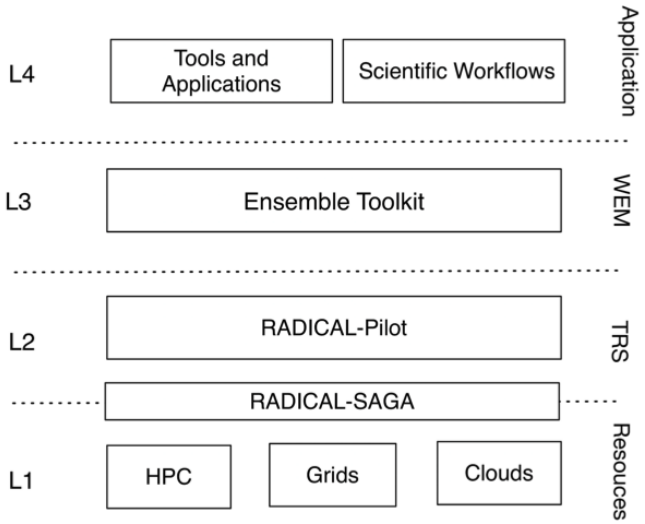

Figure 1. Primary functional levels of RCT

A container is a standard unit of software [3] that packages up code and all its dependencies so the application runs quickly and reliably from one computer environment to another. A container image is a lightweight standalone, executable package of software that includes everything needed to run an application: code, runtime, system tools, system libraries and settings.

It is lightweight since it shares the machine's OS system kernel and therefore does not require an OS per application, driving higher server efficiencies and reducing server and licensing costs.

Moreover, the image becomes a container at runtime, and said containerized software will always run the same, regardless of the infrastructure. Containers isolate software from its environment and ensure that it works uniformly despite differences for instance between development and staging.

II. DOCKER VS. SINGULARITY

With the purpose of providing a meaningful technical comparison between two of the mayor container platforms, let

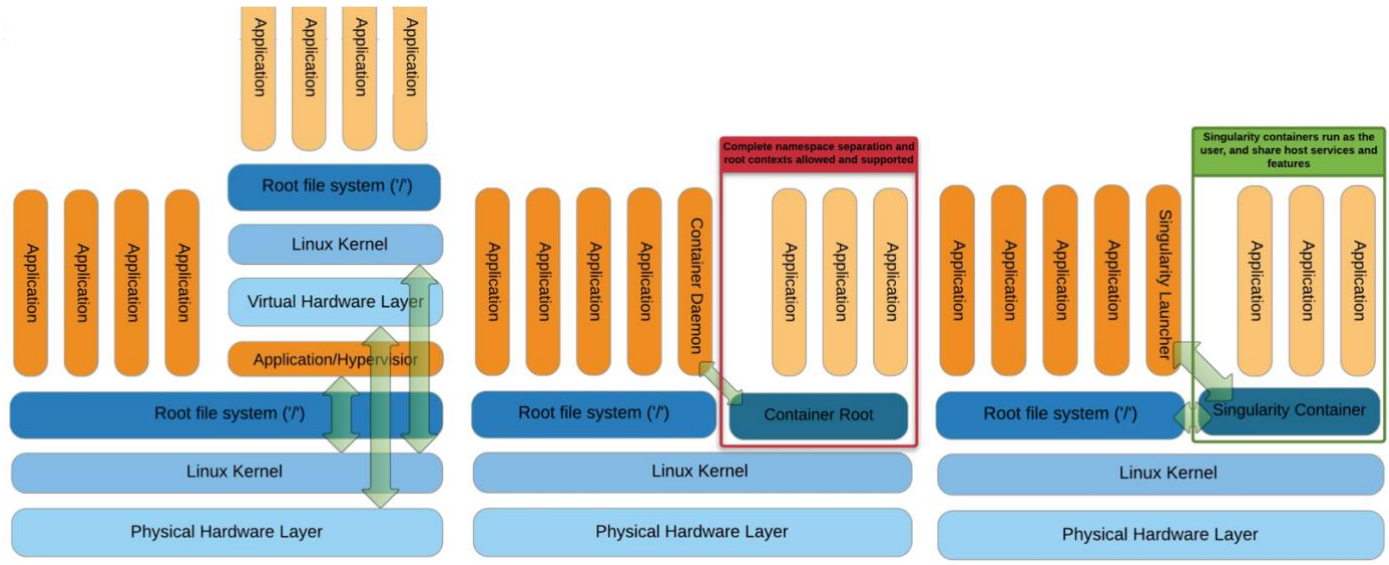

Figure 2. Left: General Virtual Machine. Middle: General Container. Right: HPC Container [4]

us define each one first.

Docker is the most well-known and utilized container platform, designed primarily for network micro-service virtualization.

It facilitates creating, maintaining and distributing container images. Containers are somewhat reproducible, easy to install, well documented and standardized.

Docker works great for local and private resources, and you can develop and share your work with others through Docker-Hub, a library of images ready to download hosted on the web. However, if you need to scale beyond your local resources, let us say, HPCs, it will not be efficient or even compatible. This is where Singularity comes in.

Singularity is a container runtime, like Docker, but it starts from a very different place. It favors integration rather than isolation, while still preserving security restrictions on the container, and providing reproducible images.

Furthermore, it enables users to have full control of their environment [7]. Singularity containers can be used to package entire scientific workflows, software and libraries, and even data. This means that you do not have to ask your cluster administrator to install anything for you, you can put it in a Singularity container and run. Also [6], it is worth noting that it is compatible with most stand-alone Docker images.

Below are some of the most important advantages [5] that puts Singularity above Docker when it comes executing on HPCs:

1. **Security**: Because of Docker daemon, a user inside the Docker container is able to obtain root access on the host. In contrast, Singularity solves this by running the container with the user's credentials.

2. **HPC Scheduler**: Users submit jobs with Central Processing Unit (CPU)/memory/time requirements. The Docker command is just an API client that talks to the docker daemon, so the resource requests and actual usages do not match. Singularity runs container processes without a daemon. They just run as child processes. In other words, Docker does not support any HPC job scheduler, but Singularity integrates seamlessly with all job schedulers including SLURM, Torque, SGE, etc.

3. **GPU support**: Docker does not support Graphic Processing Unit (GPU) natively. Nvidia Docker is a GPU-enabled Docker container, but it pre-installs various software that a user may not need. Singularity is able to support GPUs natively. Users can install whatever CUDA version and software they want on the host which can be transparently passed to Singularity.

4. **MPI support**: The Message-Passing Interface (MPI) allows the development of parallel software on HPCs. Docker does not support it natively. If a user wants to use MPI with Docker, an MPI-enabled Docker needs to be developed. If an MPI-enabled Docker is available, the network stacks such as TCP and those needed by MPI are private to the container which makes Docker containers not suitable for more complicated networks like Infiniband. In Singularity, the user's environment is shared to the container seamlessly.

To illustrate this, we can take a look at Fig. 2. Containers and Virtual Machines (VMs) have similar resource isolation and allocation benefits, but function differently due to the fact that

containers virtualize the OS instead of the hardware, thus making them more portable and efficient.

III. Experiments With RCT

Because the purpose of this performance study is to quantify Singularity's overhead, we measured the variable Total Time of Execution (*TTX*), which is the time that RCT takes to execute all the tasks. The execution overhead (*TTXovh*), expressed as a fraction of the *TTX* in the native environment, is defined as

$$TTXovh = \frac{|TTXsingularity - TTXnative|}{TTXnative} \times 100\%$$

For all the experiments, with scaled up by leaving a fixed number of cores per task.

A. *stress-ng*

The first experiment we did was on the XSEDE Bridges [10] HPC cluster, using the tool *stress-ng* [8], that places stress on a computer in various selectable ways.

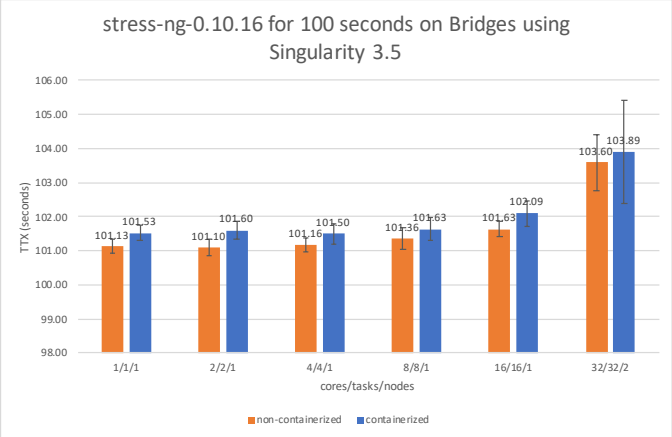

Figure 3. stress-ng on Bridges

We scaled the experiment from 1 to 32 cores/tasks. Although there was a difference in *TTX* when going from 1 node to 2 nodes, the container overhead was little to none. As Fig.3 shows, with a CPU load of 100% and a duration of 100 seconds, the maximum *TTXovh* observed was 0.5%.

B. *Hello World MPI application*

Same as before, we now went ahead and analyzed the container performance of a simple Hello World MPI executable written in C.

The challenge here is that MPI applications, just like many other HPC ones, have deep library dependencies. We have to be careful when figuring out these dependencies [7] and debug build issues. Two options are available: (i) we can install MPI inside the container, but for it to work, it has to match the one

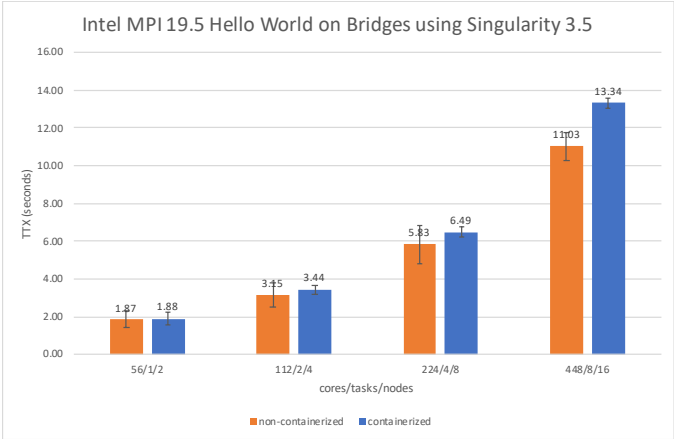

Figure 4. Intel MPI Hello World on Bridges

on the host, and (ii) we can bind the paths of the host's MPI implementation to the container so that these paths are visible to the container, but being careful with the bindings since, for example, if the OS of the host and container are different, such binding may have problems.

Since our goal is to make containers as lightweight and portable as possible, we decided to use bindings (although we experimented with both modes). There are a few MPI implementations available, mainly *Intel MPI*, *Open MPI* and *MPICH* [9, 14, 15]. On Fig. 4, you can see a comparison using Intel MPI, which is the native one on Bridges. We can see a *TTXovh* of 0.5%, 9%, 11% and 21% for 2, 4, 8 and 16 nodes respectively. We can certainly say that the container overhead is considerable when executing large-scale MPI applications via RCT.

To ensure that the previous results were independent from the HPC cluster, we performed it again but on XSEDE Comet [11] this time. However, an unexpected behavior can be appreciated on Fig. 5.

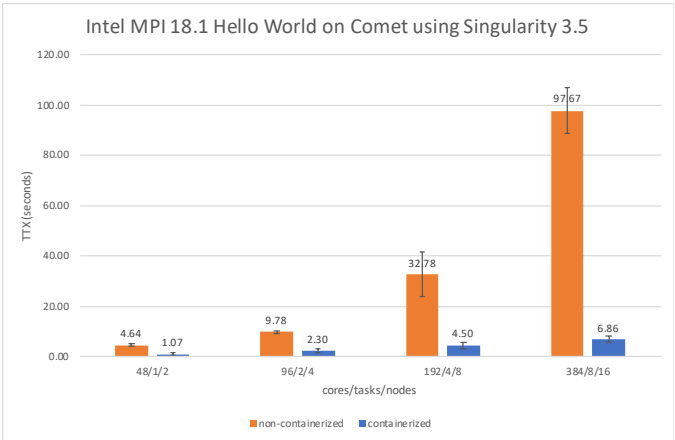

Figure 5. Intel MPI Hello World on Comet

We used the same container and executable as on Fig. 4, and somehow our execution times were considerable faster when using containers. Since Intel MPI was not the native

implementation on Comet as it is on Bridges, as a first approach at an explanation, we decided to switch to the Comet native one, which is *MVAPICH* [16], based on *MPICH*. The results can be appreciated on Fig. 6.

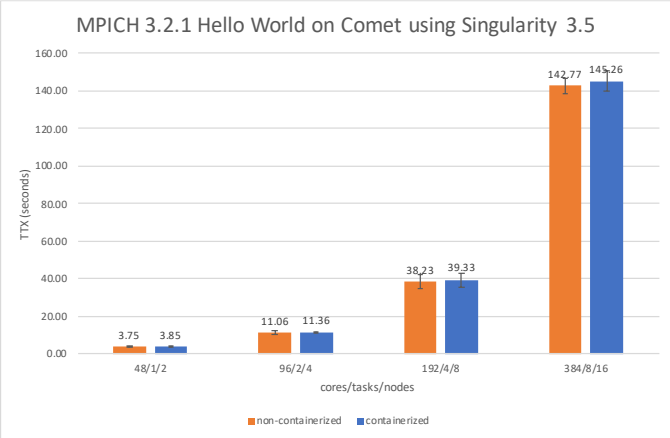

Figure 6. MPICH Hello World on Comet

As we can observe, the results were the expected. A container overhead of around 2% was achieved in all cases. Although execution times are considerably higher than on Bridges, *TTXovh* is much lower and consistent.

What happened with Intel MPI on Comet then? Our initial thought revolved around a library dependency issue and the MPI implementation/binding paths, as discussed earlier, that was not letting us set up the experiment correctly. To confirm this, we reached out to one of the members of the SDSC staff for help on this matter (Appendix C) and [17]. In summary, he provided a correctly built container with the appropriate bindings for this particular cluster. After performing the experiment again, we can see the results in Fig. 7.

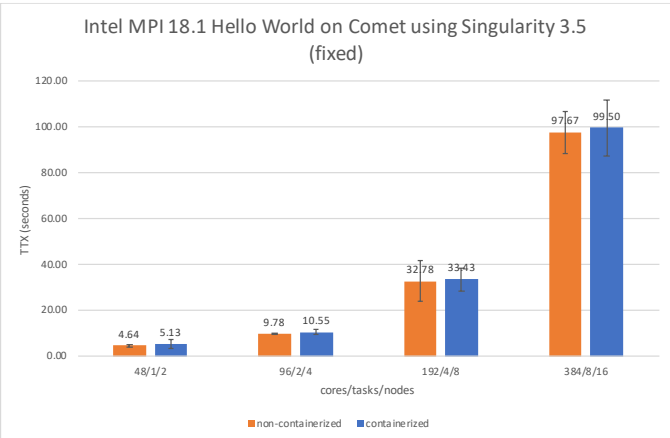

Figure 7. Intel MPI Hello World on Comet (fixed)

The observed overheads are now 10%, 8%, 2% and 2% for 2, 4, 8 and 16 nodes respectively. We see faster execution times with higher overheads.

We conclude that the MPI implementation plays an important role on the performance of our MPI applications, and depending on the HPC cluster we are executing them on, we have to be careful about what we build our containers around and specifying the correct bindings.

IV. Case Studies

A. *FACTS*

The Framework for Assessing Changes To Sea-level (FACTS) is a python library for projecting Sea-Level Rise (SLR), facilitate workflows to enhance hypothesis testing, generate large ensembles and streamline new science into SLR projections. It sits on top of EnTK.

As it is right now, automation is achieved by creating a virtual environment and installing FACTS along with its dependencies individually. This framework launches the executables for the required modules on a remote machine, like an HPC cluster, making efficient use of computing resources.

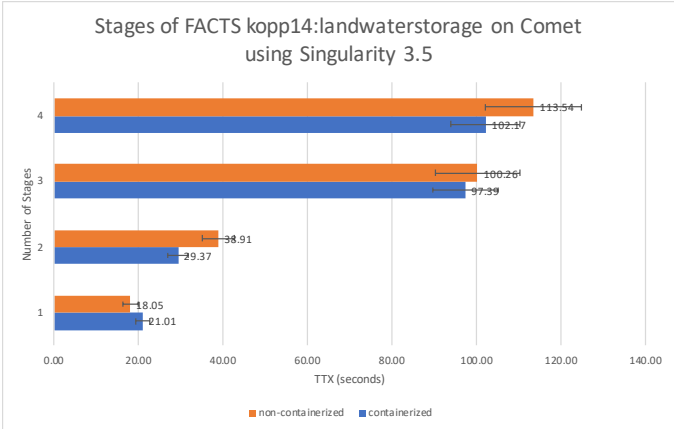

Figure 8. Stages of module "*kopp14/landwaterstorage*"

For this experiment purpose, we worked with the module set "*kopp14*", module "*landwaterstorage*". To take advantage of the benefits that Singularity can bring to FACTS, we envision the containerization at two levels: (i) at executable level and (ii) at the framework level.

The second option allow us to take FACTS apart into individual modules, completely independent from one another, with their own container each, and achieve portability and reproducibility (the end user does not have to know about anything else, no virtual environment, no dependencies, no other steps). However, this seems like an antipattern for the use of containers due to the size of the framework and its multiple functionalities. Also, FACTS requires EnTK, and this requires the installation inside a virtual environment (by design). Having a container wrapping a virtual environment takes away the purpose of having a container at all in the first place, while introducing another step for the end user to learn.

For the reasons mentioned above, we decided to start the experimentation at the executable level, and explore its potential. This way, the user still knows nothing about containers but benefits from their advantages as the difficulties to run that specific module on the cluster will be gone. The module "*kopp14/landwaterstorage*" has 4 stages: pre-processing, fit, project, post-processing. Each stage has one executable.

An aggregation of the overheads for the different stages is represented in Fig. 8. For the first stage alone, we can see the containerized execution has an overhead of 16%. For the first and second stages together, we actually appreciate a 32% improvement on *TTX*, meaning that Stage 2 executable runs faster in the container. For a workflow consisting on the first three stages, there is a 3% improvement with the containerized execution, meaning that Stage 3 has a slight overhead just like Stage 1. Finally, the complete workflow consisting of the four stages aggregated together improved by 11% when running it inside the container.

All in all, apart from the advantages we are incorporating to FACTS through Singularity, we also obtain a faster execution as an added bonus that depends mainly on the source code and how it is structured. Furthermore, we reached out for help on why we observed this behavior. If the reader is curious about it, one of the Singularity developers at Syslabs gave us a brief response. You can find it in (Appendix D).

V. Concluding Remarks

With the present work, we introduced what containers are and discussed the benefits of Singularity over Docker for HPC machines. We compared the performance of containerized/non-containerized MPI/non-MPI applications on different HPC clusters using RCT.

To conclude, it is very hard to establish a concrete answer when it comes to the container overhead. Nevertheless, we took the learned lessons and applied them to real world use cases, showing there is little to none loss in performance when containerizing our large-scale workflows with Singularity. This will allow us to make decisions based on best software engineering practices and integrate them into RCT so we can continue to support our users in terms of all the benefits containers can bring.

References

[1] M. Turilli, V. Balusabramanian, A. Merzky, I. Parasevakos, S. Jha, "Middleware building blocks for workflow systems", in *Computing in Science & Engineering (CiSE) special issue on Incorporating Scientific Workflows in Cumputer Research Processes*, pp. 62-75, July/August 2019.

[2] E. Casalicchio, V. Perciballi, "Measuring docker performance: what a mess!!!*"

[3] "What is a container?", available at https://www.docker.com/resources/what-container

[4] "Running singularity containers on comet", available at https://www.sdsc.edu/education_and_training/tutorials1/singularity_old.html

[5] R. Xu, F. Han, N. Dandaphantula, "Containerizing HPC applications with singularity", HPC Innovation Lab. October 2017

[6] R. Grandin, Y. Nanyam, "Workshop: Singularity Containers in High-Performance Computing", in High Performance Computing, Iowa State University

[7] "Singularity documentation", available at https://sylabs.io/docs/

[8] "stress-ng documentation", available at https://wiki.ubuntu.com/Kernel/Reference/stress-ng

[9] "Open MPI documentation", available at https://www.open-mpi.org/doc/

[10] "Bridges user guide", available at https://portal.xsede.org/psc-bridges

[11] "Comet user guide", available at https://www.sdsc.edu/support/user_guides/comet.html

[12] "RP documentation", available at https://radicalpilot.readthedocs.io/en/stable/

[13] "EnTK documentation", available at https://radicalentk.readthedocs.io/en/latest/

[14] "Intel MPI documentation", available at https://software.intel.com/en-us/mpi-library

[15] "MPICH documentation", available at http://www.mpich.org

[16] "MVAPICH documentation", available at http://mvapich.cse.ohio-state.edu

[17] "About comet singularity containers", available at https://www.sdsc.edu/education_and_training/tutorials1/about_comet_singularity_containers.html

[18] G. Garner, V. Balusabramanian, M. Turilli, S. Jha, R. Kopp, "A flexible computacional framework for projecting regional sea-level rise", available at https://github.com/radical-collaboration/facts

APPENDIX

A. *Documentation*

All the documentation, files, data and source code needed to elaborate this work can be found at

https://github.com/radical-group/koubbe

B. *Singularity on RP documentation page*

A step by step walkthrough of how to use Singularity with RCT on HPCs can be found at

https://radicalpilot.readthedocs.io/en/stable/

C. *Insights with Comet Research Specialist Martin Kandes*

“Hi George,

I've placed a copy of my example batch job scripts that compile and run your MPI hello, world code here [1].

As you can see in the standard output files from each job, there is really no significant difference in the performance when everything is setup correctly. i.e., I think the low runtimes you reported in the Singularity Slack channel were probably incorrect.

Anyhow, have a look at how I've set things up and let me know if you have any questions. Note, however, the container being used is not the same one you pulled from DockerHub. This one is our standard CentOS 7 Singularity container. I needed a gcc compiler in the container and the DockerHub one does not have one installed by default.

The primary difference between our container and the one from DockerHub is that the one from DockerHub is a very barebones OS. As I already mentioned, it doesn't even have a gcc compiler and supporting libraries, which were needed to support the Intel compilers and IntelMPI being mounted in the container.

Marty Kandes
SDSC User Services Group

[1] /share/apps/examples/singularity/bind-host-compilers-mpi
”

D. *Insights with Singularity Developer Dave Trudgian*

“Hello @George Koubbe,

Sometimes a container can be faster because the SIF (container image format) is a single file, and I/O on that single file is more efficient on HPC parallel filesystems than natively accessing lots of files.
”