Building Complex Software Inside Containers

Calvin Seamons
calvindseamons@lanl.gov
Los Alamos National Laboratory
High Performance Computing Division
Los Alamos, New Mexico

ABSTRACT

High performance computing (HPC) scientific applications require complex dependencies to operate. HPC centers typical support these dependencies through the use of module files, that when loaded modify the user's environment to support their application (gcc, openmpi, etc.). As user demand for HPC systems increase, along with the complexity of their applications, it becomes unrealistic to support every unique dependency. I offer containers to support the use of user defined software stacks (UDSS) as an alternative to the module filesystem. By "containerizing" Model for Prediction Across Scales (MPAS, an atmospheric simulation application), I demonstrate that it is possible to build and execute complex software within a container. Furthermore, output results are nearly identical between containers on different systems, and containers outperformed bare metal systems when running the same tests. Containers stand as a powerful alternative with equal, if not improved performance, and the ability to offer software, independent of what is offered by the HPC system.

KEYWORDS

containers, user defined software stacks, MPAS, charliecloud, unprivileged builder, bare metal

1 INTRODUCTION

HPC centers have dedicated a large amount of resources (personal and hardware) to maintain software dependencies not supplied by the Linux operating system. Offered in the form of a module file, a user loads in a dependency and their path is modified to provide access to that software installation. If a dependency doesn't exist on a system, the user must request its installation through a system administrator, or find an alternative. With application exceeding 40+ dependencies with unique installs based on MPI's and compilers it can become costly and challenging to support certain applications. Containers may offer a simplistic alternative: ship your own system. By packaging software inside a container, we can achieve the following.

- Software applications and dependencies can be built without the use of *sudo*.
- Containers can be run on different systems with nominal changes to the container.
- Nearly identical test results between containers running on machines with different hardware and operating systems.
- In the case of MPAS, containers outperformed bare metal systems (possible performance increase).
- Software that builds inside a container can execute on an HPC system regardless of what software and hardware the HPC system is running. This gives access to bleeding edge and deprecated software.

2 CONTAINER / WORKFLOW

For the containerization of MPAS I used Los Alamos National Laboratory's container builder and runtime Charliecloud. A container is a process or group of processes within a namespace. The namespace Linux Kernel feature creates the container abstraction by manipulating what resources a process or group of processes see within a given namespace. Charliecloud utilizes two of the six Linux namespaces (user and mount) to place the building of the container in a namespace with the UID equal to zero. This means during build-time the user is root (UID 0). This is used to build software applications without the use of sudo, and is known as an unprivileged builder. Once that is complete, we have a functioning filesystem image that can be distributed across an HPC system. This allows the application to execute on the HPC system freely as all software needed is installed on the container. During the execution of the container a similar process is applied. The container is manipulated into perceiving it is the sole software on the HPC system with access to kernel resources. It then proceeded with the execution of the software installed within it.

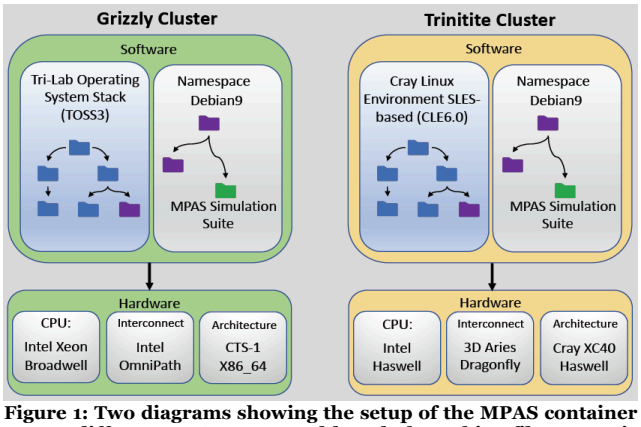

Figure 1: Two diagrams showing the setup of the MPAS container on two different HPC systems. Although the Debian file system is located lower in the toss filesystem, during runtime it thinks it is the sole OS. It also contains MPAS and can execute it upon being called.

3 Why MPAS

Model for Prediction Across Scales is open source software used by Los Alamos National Laboratory for atmospheric,

LA-UR-19-30258

ocean, land ice, and sea ice simulations. The reason it was chosen as a candidate for containerization was its relatability to other complex application used at Los Alamos. With over 40 dependencies there's no shortage of issue when it comes to building and running MPAS. Starting out, MPAS requires a newer version of gcc than what is typically offered on standard Linux distributions, gcc5.2 or greater. It also requires an MPI which is almost always needed for large scale simulation suites run in parallel. Other libraries and tools needed are Parallel-IO, Netcdf, hdf5, and NCL.

Due to the difficult nature of properly building these dependencies against each other, the success of MPAS establishes the validly of building and executing these libraries properly inside a container. MPAS serves as showcase for the possibility of containerizing other complex applications.

4 CONTAINERI ZING APPLICATIONS

The first step when it comes to containerizing an application is constructing a dockerfile. A dockerfile serves as the recipe interpreted by the container builder (Charliecoud) to construct the file system that will hold the application.

```
1  # ch-test-scope: standard
2  FROM debian:stretch
3
4  RUN    apt-get update \
5      && apt-get install -y openssh-client \
6      && rm -rf /var/lib/apt/lists/*
7
8  COPY examples/serial/hello hello
9
10 RUN touch /usr/bin/ch-ssh
```

Figure 2: A simple Dockerfile showing 3 directives (From, Run, and Copy). This particular figure is a simplistic file that pulls a debian9 image, apt-updates and runs a script that prints "hello word".

4.1 Building Applications with a Dockerfile

Dockerfiles are the industry standard when it comes to building containers. While dozens of directives exist, some important ones are featured above. **FROM** pulls in a base filesystem image that will contain the software trying to be installed. **RUN** allows the execution of commands (usually bash) and is where most of the work happens. Commands such as [wget, apt-get update, ./configure] can all happened using the run directive. Once the build process is completed, we have a working filesystem image.

4.2 Running a Container on an HPC system.

Once an image has been built via a dockerfile it's ready to run on a cluster. Using Charliecloud it takes only 4 steps.

1. Transfer the image onto a cluster.
2. Allocate nodes for your application.
3. Unpack the image onto the nodes.
4. Execute the container using charlieclouds "ch-run". This command comes with a variety of flags for customization but overall runs the container.

This is a drastic reduction in time and effort from both the user and HPC system if the application were running on bare metal.

4.3 Results of running MPAS container

MPAS was successfully able to run on different HPC clusters at Los Alamos National Lab (Trinitite and Grizzly). The atmosphere simulation, *Supercell,* produced 506 MB output files. Of the supercell output files generated, a Linux "diff" command showed that only 4 bytes differed between them. This was due to metadata such as hostname and the timestamp of the simulation start. It was also due to the location of executables being in "/usr/local" or a much deeper file path. On all tests executed the MPAS container outperformed the bare metal systems. HPC production clusters deal with so many users, a bare metal system is filled with thousands of binaries and libraries, many of which aren't pertinent to the specific application. Containers are specifically design for the software they are trying to run, as a result it is possible they execute slightly faster.

5 LIMITATIONS

Despite the success of the MPAS container there are some limitations. The first is the difficulty of containerizing software. MPAS had poor documentation and as a result tracking down developer packages and seldom used variants of libraries was a challenge. In some cases, depending on the base image one might need needs to find alternative libraries to use as there are incompatibility issues with the core program. Some applications have failed to run in containers after months of trying

Proprietary software is another challenge. As of now there is currently no support for proprietary software within containers. The means Allinea and Intel compiler will not work within a container. For some machines (e.g. Crays) post build library injections are needed to allow the container to run. Again, due to proprietary software the container cannot build these libraries and after the filesystem is built, libraries need to be injected into the container. This limits containers by adding addition steps in the build process that can be difficult to work out if not property documented.

6 IMPLICATIONS

This research shows that is it possible to build complex software inside containers. By packaging the software and all of its dependences we give code teams flexibility to run simulations across different HPC systems with nominal changes in runtime and modification to the container. This gives code teams the advantage of being able to use deprecated, or bleeding edge software on HPC systems, while also offering the possibility of improved performance. Regardless of what is on an HPC system, if it can be built in a container it can be run on a cluster.

7 FUTURE WORK

MPAS's successful containerization has shown the power of containers. To whomever is interested the next step involves simply building more software and applications within containers. Charliecloud as well as other container runtimes are all free and open sourced and it's simply to get started. By building additional software, kinks can be worked out, enhancements made, and containers can become an integral part of high-performance computing.

ACKNOWLEDGEMENTS

This research was supported by Los Alamos National Laboratories HPC Environment Group, Programming and Runtime Environments Team (PRETeam). I would like to thank my mentors Jordan Ogas and Jennifer Green.