4-2018

Making Scientific Applications Portable: Software Containers and Package Managers

Curtis Noecker
ccnoecker@csbsju.edu

Making Scientific Applications Portable: Software Containers and Package Managers

An All College Thesis

College of Saint Benedict/Saint John's University

by Curtis Noecker

April 2018

Project Title: Making Scientific Applications Portable: Software Containers and Package Managers

Approved by:

Mike Heroux

Scientist-in-Residence, Department of Computer Science

Imad Rahal

Associate Professor and Chair, Department of Computer Science

---

Noreen Herzfeld

Professor, Department of Computer Science

Jeremy Iverson

Assistant Professor, Department of Computer Science

Abstract

Scientific workflows for high-performance computing (HPC) are becoming increasingly complex. Developing a way to simplify these workflows could save many hours for both HPC users and developers, potentially eliminating any time spent managing software dependencies and experiment set-up. To accomplish this, we propose using two programs together: Docker and Spack. Docker is a container platform and Spack is a package manager designed specifically for HPC. In this paper, we show how Docker and Spack can be used to containerize the extreme-scale Scientific Software Development Kit (xSDK). Doing this makes the xSDK far more accessible to non-computer scientists and lowers time spent by developers on dependency management. Implementing a system such as this on a large scale could change the functioning of the HPC industry.

Contents

1 Introduction

Virtualization is becoming an appealing option for a growing array of computing purposes. Virtualization, meaning the creation of a virtual version of a device or resource [7], is a broad term and can apply to a variety of technologies, such as virtual hardware platforms, storage devices, and network resources. Even a simple partitioning of a hard drive would be considered a form of virtualization because you take one drive and partition it into multiple. Today, virtualization usually refers to the first of those possible applications: a mapping of a virtual guest system to a real host system. An example of this would be using Virtual Machine (VM) software such as the VMWare Horizon Client to run an instance of Red Hat Linux on a Windows machine, or vice versa. Due to the different architectures of the guest and host machines, the virtual machine must translate machine instructions, causing significant overhead to be associated with its use. [14] This kind of "traditional" virtualization has been used for years in the realm of personal computing, where the additional overhead is usually a non-issue, but has been difficult to apply to more demanding purposes, such as research and software development. Because virtual machines can provide a standardized environment, they are an attractive technology when it comes to more demanding research. However, VMs are generally unusable for these tasks because of their large overhead and how they complicate even further the already-complex series of software dependency chains required.

The high-performance computing (HPC) industry has been constantly growing and is projected to grow more than six percent annually over the next five years. [9] However, as HPC grows, finding a solution to the problem of environment standardization becomes ever more important. Primary uses of HPC are often scientific analyses of large data sets, requiring scientific software workflows with complex sets of dependencies. [13] Maintaining these workflows requires significant time and resources as well as an in-depth knowledge of

the HPC itself. A new virtualization technology, containerization, has been quickly gaining attention from HPC researchers as a potential solution to these problems. This thesis aims to show how a fast-growing container program, Docker, and a specialized software package management tool, Spack, can be used in tandem to create a container with a clean install of the extreme-scale Scientific Software Development Kit, solving the issues of environment standardization and increasingly complex scientific software workflows.

2 Background

2.1 Docker

Docker, and application-level containers in general, are a form of operating system-level virtualization. This form of virtualization is a server virtualization where the operating system's kernel is divided into multiple user space instances, each one separate from the others. These instances, called containers, are completely isolated from the others and are, in a way, a form of virtual machine. Users operating within these containers experience operations as if they are working on their own server. Docker adds a level of abstraction to this operating system-level virtualization and runs "application-level" containers on top of these user instances.

With Docker, the abstraction is the application itself. Docker allows for applications to be run by 1 or many containers, where many small services (provided by each individual container) combine to create an application. [5] In this way, Docker functions similarly to server operating system-level virtualization, just at a higher level of abstraction; each container operates independently of the others in its own standard environment with each container together forming the application. Isolation features such as Linux *cgroups* are what allow Docker to simultaneously run isolated and independent application containers.

[12] *cgroups*, or control groups, is a feature of the Linux kernel that allows the user, in this case Docker, to allocate resources, such as time, memory, or bandwidth, among groups of tasks running on a system. [6]

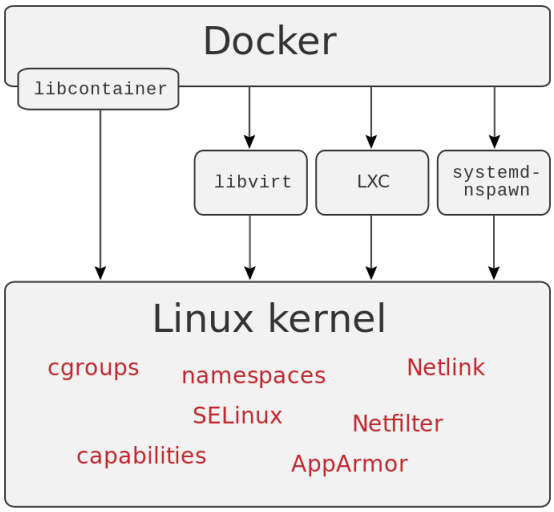

Figure 1: This diagram shows the various execution environments compatible with Docker after update 0.9. Also shown are Linux operating system-level container APIs used by Docker containers. [11]

Docker began in France as an internal project of dotCloud, a platform as a service company. First released to the public in March of 2013, Docker quickly gained traction and expanded into its own corporation. Docker was originally built using Linux Containers (LXC). Using the LXC led to many issues, such as being far from user friendly and having major security concerns. Because of these issues, a year after its original release, Docker updated to version 0.9 and changed changed its entire infrastructure. Instead of relying exclusively on LXC, Docker developed its own execution environment: *libcontainer*. The development of libcontainer meant Docker was now a complete package and did not rely on outside technology. As seen in Figure 1, although Docker developed libcontainer, it is still compatible with outside interfaces such as LXC and others, although libcontainer is the default. [11] This update also cleared the way for Docker to be made available for non-Linux platforms. [15] Today, Docker releases are available for IaaS Clouds in addition to versions

for Mac and Windows. In 2015, Docker partnered with large companies such as Microsoft, Amazon, and Google to begin the Open Container Initiative (OCI). The OCI's stated goal is to create "open industry standards around container formats and runtime." [3] Now an open source project on GitHub, Docker continues to grow as its product improves.

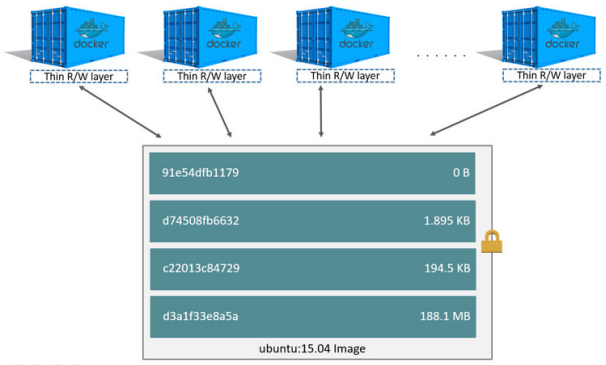

Figure 2: In this figure, the different layers of a Docker Image are evident. Each top container "Read/Write layer" represents a running container. The box underneath the running containers represents the Ubuntu image that is being used to run the containers. Each layer within the image is one command in its Dockerfile. For example, layer c22013c84729 adds data from the Docker client's current directory with a Dockerfile COPY command. [1]

Docker containers are run from files called "images". A Docker image is an inert file which represents a container, as if it is a "snapshot" of the container. When run, images will produce a container, much like how a class produces an object. The relationship from image to container is very similar to that of class to object; an image is a class and the container is the object created by that class. An image is a file, usually stored in a registry like the Docker Hub (registry.hub.docker.com) and built from what is called a Dockerfile. [1] A Docker image is built in layers, where each layer represents an instruction in its Dockerfile. Each layer in an image is nothing but the set of differences from the layer before it. When a container is created, a new writable layer is added on top of the underlying image layers, called the "container layer". This system, seen in Figure 2, keeps image files from growing too large and enables different, simultaneously running, containers which all access the same

image to maintain their own data state.

2.2 Spack

Spack (Supercomputer Package manager) is a tool that is meant to benefit HPC workflows. Spack, like Docker, is a very recent technology. Originally developed in 2013 at Lawrence Livermore National Laboratory (LLNL) by Todd Gamblin, the tool was released publicly in 2015. To manage the needs of a diverse set of applications, system administrators at LLNL spent "countless hours" dealing with build and deployment issues. [10] Spack helps deal with a major problem in HPC: managing software dependencies.

Spack is a package management tool designed specifically for large supercomputing centers. Spack has a variety of unique features intended to make managing scientific applications simpler and easier. First, Spack is non-destructive. Spack is designed to support multiple versions and configurations of the same software on a single system. Scientific applications require specific versions of compilers, MPI, and other dependency libraries, making it infeasible to use a single, standard software stack. A second novel feature of Spack is its build methodology. Given a command to install a package (and it could be any version of that package), Spack will first recursively install each of the dependencies for that package that are not already installed on the machine and automatically locate the dependencies which are already installed. Significantly, Spack does this regardless of its environment. In a case where custom dependencies are needed, Spack can be easily configured to install custom builds.

2.3 The xSDK

The Extreme-scale Scientific Software Development Kit (xSDK) is a collection of related and complementary software elements developed by separate teams throughout the HPC

community. The stated vision of the xSDK is to provide infrastructure and interoperability for this collection of software. The collection of software is meant to provide the "building blocks, tools, models, processes, and related artifacts for rapid and efficient development of high-quality applications." [2]

As of xSDK-0.3.0, released in November of 2017, the xSDK contains seven numerical libraries and two application packages. When it was first released in April 2016, the xSDK contained only four numerical software libraries and one application component: hypre, PETSc, SuperLU and Trilinos, and Alquimia.

- Hypre provides high-performance preconditioners and solvers for the solution of large, sparse linear systems on massively parallel computers.
- PETSc is a suite of data structures with routines for the scalable solutions of applications modeled by partial differential equations.
- SuperLU is a more general-purpose library, meant for the solution of large, sparse, nonsymmetric systems of linear equations.
- The Trilinos Project is organized into 66 separate packages, each with a different focus. Overall, it is an effort to develop algorithms and technologies within an object-oriented framework for the solution of large-scale and complex multiphysics problems. Some of the packages within Trilinos include linear and nonlinear solvers, eigensolvers, and graph partitioners, among others.
- Alquimia is a biogeochemistry API and wrapper library that aims to provide a unified interface to existing geochemsitry engines such as CrunchFlow or PFLOTRAN. [4]

Beginning with xSDK-0.2.0, released in February of 2017, a version of the xSDK was made compatible with the Spack installer. With the 0.3.0 update, Spack has been made the default way to download and install the xSDK.

3 General Benefits of Using Docker and Spack

3.1 Benefits of Docker and Container Technology

Benefits of Docker can be seen mainly in two areas: development and distribution.

- Benefits of Docker for development
  - **Standard execution environment**

    When developing scientific applications, the execution environment can sometimes have a large effect on the results of a program. For this case, Docker containers are the perfect test environment. As Docker is a complete environment, applications will always run the same way regardless of the container's host environment. A developer can test code inside a container, and when it completes successfully, be confident that pushing their code will not break the application. Testing code in a native environment carries many risks; it is not uncommon for new code to work perfectly on a personal system but bring a whole system down when pushed to the master branch. Docker containers solve this problem.
  - **Bug reproduction**

    In addition, containers are useful for bug reproduction. By maintaining a number of standardized images for issue handling, bugs can be easily reproduced in a variety of different environments. You could say that VMs can be used for the same purpose: a clean install of a VM will also function as a standard environment for either testing or bug reproduction. However, containers are much smaller in size than VMs and, because of how images are made of layers, can come with software such as the xSDK pre-installed.
  - **Image size**

    When comparing to a VM, Docker containers are able to have smaller size because

of how they (a) are built of layers, and (b) do not require a guest operating system. Both VMs and Docker containers include the application (i.e., the xSDK), the libraries, and other required files. The difference is that VMs require a guest operating system (and the space and cost that comes with it), whereas containers do not. [16]

- Benefits of Docker for distribution
  - **Simpler and easier distribution**

    Docker provides much easier distribution of software due to simplified access on the user's end. HPC end-users are oftentimes not computer scientists and getting software installed correctly so that their experiment will run correctly is a big concern. Accessing software using Docker makes the process straightforward thanks to the ease with which image registries such as Docker Hub or Quay.io can be used. There is no need for downloading, building, and installing software if a developer's image has been shared to the registry; a user can pull any Docker images and have near-instant access to the software enclosed.
  - **Distribution of software and environment**

    By using Docker to distribute their software, a developer not only distributes their software, but it distributes the entire environment as well. This lets users avoid any and all environment-related issues such as what environment settings this specific HPC might have that would break their program. These environment-related issues are problems that any non-computer scientist would have a lot of trouble in resolving. However, if a developer distributes their software through Docker, users can avoid this problem.
  - **Minimal effects on performance**

    Running a container on any machine has been shown to have little to no effect on

performance, strengthening the case for using containerized software. [8]

3.2 Benefits of Spack

There are five main features of Spack that set it apart from other modern HPC package managers. Each one contributes to making package management a simpler, more automatic process for users. Written in pure Python, each of these unique features are meant to help meet this goal. The first of these features is the ability to easily compose packages that are explicitly parameterized by version, platform, compiler, options, and dependencies. Spack packages, which function as Python scripts, are a class which extends a generic *Package* base class. *Package* implements most of the build process, leaving just an *install* method to user implementation to handle specifics of particular packages.

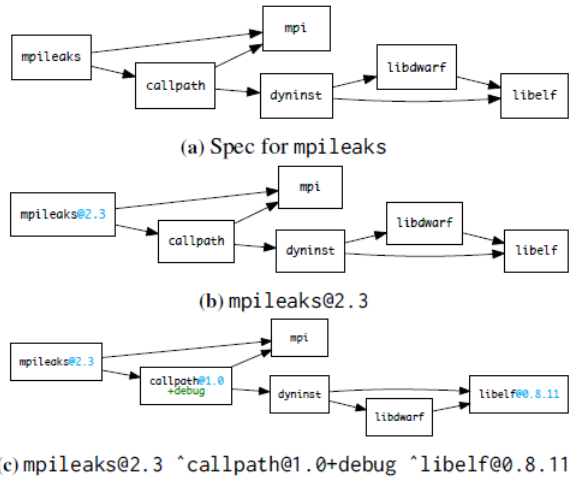

Figure 3: A diagram that shows the Spack spec syntax dependency directed acyclic graphs for different *mpileaks* installs. [10]

A second important feature of Spack is its novel and recursive spec syntax for dependency graphs and constraints. In Spack, *spec* refers to a single build configuration. Spack's specs recursively inspect the class definitions for each dependency of a package and construct a graph of their relationships, resulting in a directed, acyclic graph (DAG) as in Figure 3. This

is how Spack maintains many configurations of the same package: no two configurations would ever have the same build DAG. At the same time, a user can specify versions for dependencies in addition to the package itself, such as in Figure 3c, because of how the spec syntax is recursively defined. [10]

The three other central, unique, features of Spack are virtual dependencies, a new concretization process, and a new installation environment that uses compiler wrappers. The virtual dependency feature works around the common issue of there being multiple libraries that share a common interface and which can be interchanged within a build. An example of this would be the many implementatins of MPI, including MPICH and OpenMPI. Spack's virtual dependencies represent a library interface instead of an actual implementation. This allows users to select the implementation that they desire at build time. Spack's concretization process takes the abstract spec DAG of a package, and follows a process to ensure that three conditions are met: 1) no package in the DAG has missing dependencies, 2) no packages are virtual, and 3) all parameters are set for each package in the DAG. The unique installation environment of Spack is meant to build a consistent HPC stack and achieve reproducible builds across machines. [10]

4 The xSDK Docker Container

The appeal of Docker and other application container technologies, such as Singularity, an HPC-specialized container technology, have been expanding rapidly. This appeal extends to scientific software development and computational research. The xSDK, while not being an application itself, is meant for the creation of scientific packages. Containerizing a Spack installation of the xSDK takes full advantage of the benefits that both Docker and Spack have to offer.

4.1 The Dockerfile

Creating this Docker container involves building from a straightforward Dockerfile (Appendix A.1). The xSDK container is built from a base CentOS image, which contains nothing except a very bare-bones version of the CentOS. The Dockerfile then specifies which dependency packages need to be installed separately from the xSDK both so Spack can be installed and so Spack can install the xSDK. These packages include git, gcc, make, and patch. Spack is supposed to be able to install all missing dependencies for a package automatically, but these packages that need to be installed separately are so essential to all systems that Spack assumes they are always present and does not account for them when building the DAG for a package. These separate installations are done using the default CentOS installer yum. The next step is to clone Spack from its git repository and install its environment modules. Once Docker has done this, it uses bash to complete the xSDK installation, invoking the "spack install xsdk" command to install the xSDK and its dependencies and then loading the installed xSDK. This build process has now created a Docker image that can be run as a container with the xSDK installed.

4.2 Accessing the xSDK Container

Accessing and running the xSDK Container is a simple process if Docker has already been installed on your machine. First, use the `docker pull` command to pull the `xsdk/xsdk` image from the Docker registry. With the image downloaded, it is easy to run the container with a simple command: `docker run -it (--name=xSDK) xsdk/xsdk /bin/bash`. The `--name` tag is optional and only provides a name for the container created. The `run` command is a docker command which takes the image listed (in this case `xsdk/xsdk`) and executes it to create a running container. The `-it` options together allocate a tty for the container process. This is necessary to create an interactive shell such as bash. The final `/bin/bash`

command is a default command that is executed when the container is created. With these two simple steps of pulling the image and running it, any user has full access to the full contents of the xSDK.

4.3 Specific Advantages of a Containerized xSDK

As discussed above, containerizing applications and software like the xSDK has benefits for both development and distribution. The first and main advantage of having the xSDK available from a Docker container is the much greater portability that the software has when it is packaged this way. Advantages for the distribution of the xSDK are generally abstraction-related: potential users of the xSDK do not need to worry about their native environments and obtaining the xSDK becomes simpler. The native environment of an xSDK user no longer matters; the only requirement is that it can run a version of Docker. Additionally, when using the xSDK to develop their own applications, users do not need to concern themselves that errors might be unique to their machine. The other main benefit is that obtaining this version of the xSDK is far more straightforward. Users do not need to have any part in the installation of the xSDK. With Docker installed, users can pull the 2.22GB xSDK image from the Docker Hub and have immediate access to the software enclosed. The benefits for development are similar to those for distribution and center mainly on portability and simplified debugging. Similar to its users, developers of the xSDK do not need to think that errors may be specific to their machine and can be confident that pushing changes to the master branch will not break the software.

4.4 Specific Advantages to using Spack

Using Docker and Spack together greatly increases the accessibility and ease of use of the complex scientific packages which make up the xSDK. Each one of these packages has their

own complex set of dependencies that would be very difficult to manage and maintain on their own, so the xSDK is installed inside the container using Spack. Installing the xSDK using Spack has made the installation process extremely straightforward. In addition to installing the xSDK, Spack first installs all necessary dependencies of the xSDK. Spack can analyze the environment it is working in and recognize what lower-level packages are needed before installing the xSDK – removing any dependency maintenance within the container that would otherwise be required.

5 Conclusion

Docker and other container technologies continue to grow at a rapid rate. A very recent technology, the future potential of containers continues to increase. The variety of purposes a container can fulfill while having little to no effect on performance shows that it is a technology that will continue to have a growing impact on both the general world of computers and on HPC. Containers, especially when coupled with a smart package management tool such as Spack, have the potential to reshape how the HPC industry functions, adding multiple levels of abstraction and making the process much more straightforward to obtain and use the xSDK. Using such a portable and accessible tool will allow a much larger consumer base to access high-level scientific software such as the xSDK.

Barriers remain to widespread adoption of containers. The largest is the steep learning curve associated with learning Docker. Obtaining and running a existing image, such as the xSDK, is straightforward if Docker is already installed on the machine. But learning the subtleties of how to create, operate and maintain your own containers is a long process. There is no doubt that becoming comfortable deploying applications using Docker requires significant time and effort.

5.1 Future Research

The first, clear next step with this research is to run comprehensive tests on the containerized xSDK. These tests would make sure all functionalities of each package are working as intended. They should also be compared to tests run natively to evaluate performance. If performance is significantly worse for the xSDK when running inside a container, its benefits would be reduced to only error testing and increased accessibility. Another step that could be taken would be to produce a collection of xSDK containers, each containing a different build version of the xSDK. This would make the xSDK available to more users who might not necessarily want the default xSDK installation.

References

[1] About images, containers, and storage drivers.

[2] xsdk:extreme-scale scientific software development kit.

[3] About - open containers initiative. 2016.

[4] alquimia-dev. 2016.

[5] *Docker for the Virtualization Admin*. Docker, 2016.

[6] Chapter 1. introduction to control groups (cgroups). 2018.

[7] Vangie Beal. virtualization. 2018.

[8] Sean J. Deal. Hpc made easy: Using docker to distribute and test trilinos.

[9] Michael Feldman. Hpc market hits record revenues in 2016, looks ahead to double-digit growth in ai. 2017.

[10] Todd Gamblin, Matthew LeGendre, Michael R. Collette, Gregory L. Lee, Adam Moody, Bronis R. de Supinski, and Scott Futral. The spack package manager: Bringing order to hpc software chaos. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '15, pages 40:1–40:12, New York, NY, USA, 2015. ACM.

[11] Solomon Hykes. Docker 0.9: Introducing execution drivers and libcontainer. 2014.

[12] Bill Kleyman. Understanding application containers and os-level virtualization. 2015.

[13] Gonzalo P. Rodrigo, Erik Elmroth, Per-Olov Östberg, and Lavanya Ramakrishnan. Enabling workflow-aware scheduling on hpc systems. In *Proceedings of the 26th Inter-*

national Symposium on High-Performance Parallel and Distributed Computing, HPDC '17, pages 3–14, New York, NY, USA, 2017. ACM.

[14] James E. Smith and Ravi Nair. Chapter five - high-level language virtual machine architecture. In James E. Smith and Ravi Nair, editors, *Virtual Machines*, The Morgan Kaufmann Series in Computer Architecture and Design, pages 221 – 279. Morgan Kaufmann, Burlington, 2005.

[15] Chris Swan. Drocker drops lxc as default execution environment. 2014.

[16] Nestify Team. Why you should start using docker today?, 2017.

A Appendix

A.1 The xSDK Dockerfile

```
##############################################################
# Dockerfile containing required packages for building Spack and xSDK using Spack
# Based on CentOS image
##############################################################
FROM centos
MAINTAINER ccnoecker
# Installing base packages which Spack assumes to be installed on any machine but that the
# very simple CentOS image does not contain.
RUN yum install -y \
git \
gcc \
make \
cmake \
gcc-c++ \
patch \
gcc-gfortran; \
# Cloning Spack from Github and using it to install environment modules and bzip2.
git clone https://github.com/llnl/spack.git /usr/spack; \
./usr/spack/bin/spack install environment-modules \
./usr/spack/bin/spack install bzip2
# Using Spack to setup environment and install and load the xSDK.
RUN /bin/bash -c 'source './usr/spack/bin/spack location -i environment-modules'/Modules/init/bash;\
. /usr/spack/share/spack/setup-env.sh; \
```

```
spack load bzip2; \
spack install xsdk; \
spack load -r xsdk'
EXPOSE 80
CMD ["/usr/sbin/init"]
```