Study of a Docker use-case for HEP

Jim Kowalkowski, Adam Lyon, and Marc Paterno

Revision 1

Contents

1 Introduction

Over the past few years, container technology has become increasingly promising as a means to seamlessly make our software available across a wider range of platforms. In December 2015, we decided to put together a set of docker images that serve as a demonstration of this container technology for managing a run-time environment for *art*-related software projects, and also serve as a set of test cases for evaluation of performance.

Docker[1] containers provide a way to "wrap up a piece of software in a complete filesystem that contains everything it needs to run". In combination with Shifter[2], such containers provide a way to run software developed and deployed on "typical" HEP platforms (such as SLF 6, in common use at Fermilab and on OSG platforms) on HPC facilities at NERSC.

Docker containers provide a means of delivering software that can be run on a variety of hosts without needing to be compiled specially for each OS to be supported. This could substantially reduce the effort required to create and validate a new release, since one build could be suitable for use on both grid machines (both FermiGrid and OSG) as well as any machine capable of running the Docker container. In addition, docker containers may provide for a quick and easy way for users to install and use a software release in a standardized environment.

This report contains the results and status of this demonstration and evaluation.

1.1 Goals

The overall goals of this project were the following:

- To generate a container with a standard release of *art*, delivered to the container through UPS, with the run-time environment completely established when the image is booted.
- To deploy the container through Dockerhub[3] onto Cori[4], and use Shifter through the Cori SLURM batch system to run *art* using standard simulation and reconstruction workflows.
- To generate a container with an *art* release re-installed directly into the standard basic Linux directory structure `/usr`. This effectively removes UPS and makes the entire release (including externals) part of the platform. This part was not tackled in this initial exercise.

An initial set of goals and tasks was also established[6].

The specific container technology exploration goals were:

- to utilize the layer features of containers when building images,
- to utilize the Docker push and pull facilities for registering and accessing built images, and
- to understand how to utilize docker images on Cori.

1.2 Rationale

Some of the advantages of moving towards container technology have already been mentioned in the introduction. These include

- ease of deployment onto additional Linux variants for operations,
- reduced validation requirements for new Linux variants,
- ease of establishing a run-time and development environment for new users.

Our current software distribution system (UPS) exists in large part to support the installation of multiple versions and variants of each required software product, allowing the user of UPS to select which set of products are to be available in a given shell, while providing a fairly strong guarantee that the set of software products being used are binary-compatible. One significant drawback of the UPS system is the degree of expertise required to package third-party software for delivery through UPS.

In our UPS releases, we deliver a compiler suite because we must assure we have a sufficiently new compiler version on all supported platforms. Since the system compiler on some systems is too old, we cannot use system compilers.

There may be noteworthy advantages to re-installation directly into the standard Linux `/usr` directory. By using a base image built from a sufficiently current operating system, we can assure that the system compiler is sufficiently new. For example, at the time of this writing the Ubuntu 16.04 release, with a system compiler of GCC v5.3.1, is available. If we are using the system compiler, we can also then use packages built and distributed by the usual system installation tools.

This means no special handling is required to have consistency in all the installed software and no requirement to have more than one compiler installed and made available through UPS. This applied equally to the Python interpreter and environment. Many of the Python tools and libraries demand a standard installation to work well together, and establishing a image with python as a platform tool (as opposed to a UPS external package) will again help with consistency and easy of adding new libraries and tools.

1.3 Scope

The main focus is the *art* run-time environment, installed using a standard SLF 6 release on an x86_64 platform, and tested on NERSC's Cori. The target was *art* version 1_18 (with ROOT 6) and version 1_17 (ROOT 5), each with qualifier e9 (which means they were built with GCC version 4.9.3, using the C++ 2014 language standard).

The testing here does not include operating through production grid facilities and services, and does not include use of CVMFS.

We did explore how the several of the LHC experiments (ALICE, ATLAS, and CMS) are using Shifter and CVMFS to test software release at NERSC. We met with CMS to find out the status of testing container technology at NERSC.

1.4 Terminology

Container: "Docker containers wrap up a piece of software in a complete filesystem that contains everything it needs to run: code, runtime, system tools, system libraries anything you can install on a server. This guarantees that it will always run the same, regardless of the environment it is running in." — from `https://www.docker.com/what-docker`.

Image: "A Docker image is a read-only template. For example, an image could contain an Ubuntu operating system with Apache and your web application installed. Images are used to create Docker containers. Docker provides a simple way to build new images or update existing images, or you can download Docker images that other people have already created. Docker images are the build component of Docker." — from `https://docs.docker.com/engine/introduction/understanding-docker`.

Shifter: "Shifter is a prototype implementation that NERSC is developing and experimenting with as a scalable way of deploying containers in an HPC environment. " — from `http://www.nersc.gov/research-and-development/user-defined-images`.

Dockerhub: is a free repository for Docker images.

btrfs: "Btrfs is a new copy on write (CoW) filesystem for Linux aimed at implementing advanced features while focusing on fault tolerance, repair and easy administration." — from `https://btrfs.wiki.kernel.org`.

2 Summary

We produced a series of Docker images, layered to reflect the dependencies of the software products that each contains. The layers start at a basic SLF-compatible OS, building up toward *art*, then larsoft, and ending at an experiment-specific release. The two final images were published on DockerHub for testing: one containing a recent MicroBooNE uboonecode release (v4_31_00) and one containing a ROOT 6 *art* release configured for development with a study package (v1_18_03). Both of these images were deployed on Cori at NERSC.

We ran two test applications on Cori under Shifter using the SLURM batch system and on our local development server (native and under Docker). The applications were: a standard MicroBooNE simulation (uboonecode image) and a simple I/O and overhead exercise (*art* study image). This paper shows consistent results amongst all three of these environments for CPU and write performance.

The Cori tests demonstrate reading configuration from and writing simulation output to the global file system using the scratch area under our user accounts. This exercises an important feature of writing directly from the image running on a compute node to the externally mounted global file system.

The procedure developed to complete this demonstration is a good start for moving towards general use of Docker facilities for experiment software releases. All of the work we have done is available in DockerHub, GitHub, and Fermilab's Redmine.

3 Methods, Assumptions, and Procedures

3.1 Machine configuration

The following machines were involved in this project:

- woof.fnal.gov (SLF 6): the initial primary build site for images.
- cori.nersc.gov (Cray SuSE EL7 with v3 kernel).
- VM on Mac laptop (Centos 7).
- VM on woof (Centos 6.7).

3.1.1 Installing Docker for RHEL 7 (or equivalent) OS

For Centos7, we used the procedure at https://docs.docker.com/engine/installation/centos/. The major parts of the procedure are

1. Make sure installed packages are up-to-date: `yum update`.
2. Add the Docker repository for yum to use:

```
sudo cat > /etc/yum.repos.d/docker.repo <<-EOF
[dockerrepo]
name=Docker Repository
baseurl=https://yum.dockerproject.org/repo/main/centos/7/
enabled=1
gpgcheck=1
gpgkey=https://yum.dockerproject.org/gpg
EOF
```

3. Install the Docker client and daemon: `yum install docker-engine`.
4. Start the Docker daemon: `service docker start`
5. Make sure the Docker daemon will be restarted on reboot: `chkconfig docker on`
6. Add the users who will use Docker to the `docker` group: `usermod -a -G docker <user>`.

3.1.2 Installing Docker for RHEL 6 (or equivalent) OS

For a RHEL 6 installation, more work is required.[1] Furthermore, RHEL 6.5 or newer is required.

The docker system was established on woof and the Centos6.7 VM using the installation procedure at `https://docs.docker.com/v1.5/installation/rhel/#red-hat-enterprise-linux-6.5-installation`, with some additions.

The installation commands require root privileges; we assume the user is logged in as root. Here are the basic steps:

1. Make sure EPEL is included in the list of accessible yum repositories.
2. Make sure the most recent list of yum packages is installed: `yum update`.
3. Remove the unrelated *docker* product, which collides with the Docker we are trying to install: `yum -y remove docker`.
4. Install the Docker product: `yum install docker-io`. Note this is a different RPM name than is used under RHEL 7.

The RHEL 6 default installation of filesystems and Docker device manager (`devicemapper`) can not handle Docker images larger than 10 GB. This is insufficient to allow building the MicroBooNE image.

To handle this, we had to switch to using btrfs[7] for the filesystem that docker uses on woof.fnal.gov. To do this, we used the following prescription.

1. Format a new disk partition using btrfs,
2. Shut down docker and remove `/var/lib/docker`
3. Mount the new btrfs partition and link it to `/var/lib/docker`
4. Because the Docker daemon from the RPM above was not built with btrfs support, we then had to install a new Docker image directly from the Docker site.
   The docker image can be installed with:

   ```
   > wget https://get.docker.com/builds/Linux/x86\_64/docker-1.7.1 \
       -O /usr/bin/docker
   > chmod +x /usr/bin/docker
   ```

   It is also necessary to edit `/etc/sysconfig/docker`. The line that sets `other_args` should specify that the btrfs filesystem should be used:

   ```
   other_args="-s btrfs"
   ```

5. Start the Docker daemon: `service docker start`.
6. Make sure the Docker daemon starts on reboot: `chkconfig docker on`.

3.2 Building images

Each Docker image used in this study was created through instructions in a Dockerfile, which is the standard mechanism for obtaining repeatable builds of Docker images.

The command to build a Docker image is:

[1]We note that a very old version of Docker is the newest version that is available for RHEL 6.x. As of this writing, the current version of Docker is 1.10; the newest available for RHEL 6.x is Docker 1.7.1.

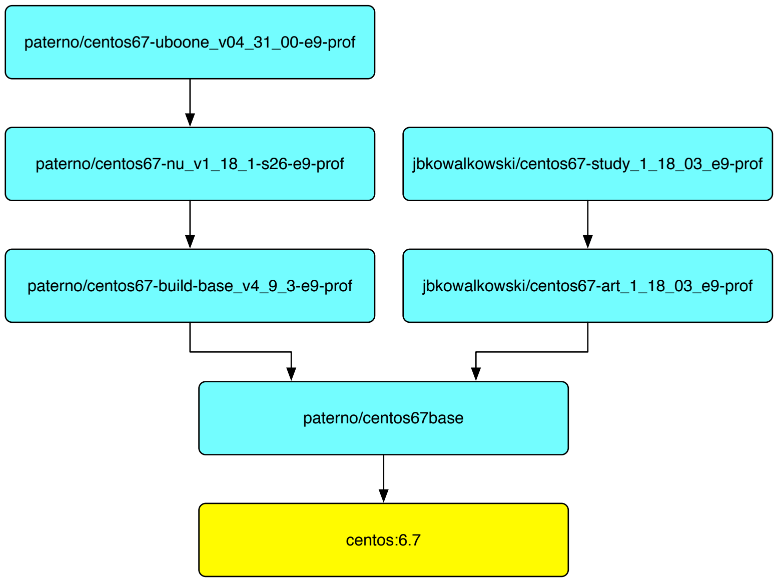
paterno/centos67-uboone_v04_31_00-e9-prof
paterno/centos67-nu_v1_18_1-s26-e9-prof
jbkowalkowski/centos67-study_1_18_03_e9-prof
paterno/centos67-build-base_v4_9_3-e9-prof
jbkowalkowski/centos67-art_1_18_03_e9-prof
paterno/centos67base
centos:6.7

Figure 1: The layering of the Docker images used in the performance study. The names are the Dockerhub identification of the images. The yellow box shows the official CentOS 6.7 image, provided by the CentOS project. The images denoted by the blue boxes were produced as part of this study.

```
> docker build -t <tag> <path-to-Dockerfile>
```

Images can be created on any host system that supports Docker; the resulting images can be loaded and run as containers from any other system that supports Docker.

Our strategy for building images, for this study, was to create a layer corresponding to each UPS product bundle in our usual software delivery system. Figure 1 shows the various layers that we produced. The CentOS 6.7 image is one provided by the CentOS Project.

We started with the CentOS 6.7 image because it is compatible with SLF 6, which is the main OS under which the SCD supports software. This allowed us to install already-built UPS packages into the Docker image. The Dockerfile for building this image is in listing 3.1. This creates an image that gives us all the products needed for building all of our own software, and for all the third-party products we package through UPS.

```
FROM centos:6.7
MAINTAINER XXXXX
LABEL Vendor="FNAL"
LABEL License="BSD"
RUN yum install -y gcc
RUN yum install -y redhat-lsb-core
RUN yum install -y atk-devel apr-devel asciidoc cairo-devel
   libconfuse-devel libcurl-devel fontconfig-devel freetype-devel
   glib2-devel gtk2-devel krb5-devel libICE-devel libSM-devel libX11-
   devel libXext-devel libXft-devel libXi-devel libXrender-devel
```

```
    libXt-devel libpng-devel libstdc++-devel mesa-libGL-devel mesa-
    libGLU-devel ncurses-devel nss-devel openssl-devel openldap-devel
    pango-devel qt-devel swig texinfo xmlto zlib-devel readline-devel
    expat-devel libXpm-devel libXmu-devel bzip2-devel texinfo tk-devel
     tcl-devel gcc-c++ libgcc.i686 glibc-devel.i686 libstdc++.i686
RUN yum -y install epel-release
RUN yum install -y libconfuse-devel
CMD ["/bin/bash", "-l"]
```

Listing 3.1: Dockerfile used to construct our base image.

Listing 3.2 shows the Dockerfile used to build the centos67-build_base image. This image contains the suite of compilers and related tools with we we build the rest of the software we deliver. We use `/etc/profile` to establish the appropriate UPS products as active for all login shells. Thus the user, upon starting a container based on this image (or on one built upon it) does not need to set up the UPS environment, not the set of compilers to be used for development work.

```
FROM paterno/centos67base
MAINTAINER Marc Paterno paterno@fnal.gov
ENV REFRESHED_AT 2015-12-28
LABEL Vendor="FNAL"
LABEL License="BSD"
RUN mkdir -p /products \
  && cd /products \
  && curl -O http://scisoft.fnal.gov/scisoft/bundles/tools/
     pullProducts \
  && chmod u+x pullProducts \
  && ./pullProducts -r $PWD slf6 build_base-v4_9_3 e9 prof

COPY build_base_setup.sh /etc/build_base_setup.sh
RUN echo 'source /etc/build_base_setup.sh' >> /etc/profile

# Default command
CMD ["/bin/bash", "-l"]
```

Listing 3.2: Dockerfile used to construct the centos67-build_base image.

Listing 3.3 shows the Dockerfile that was used to construct the *art* v1.18 image. This image contains an installation of version 1.18 of the *art* framework. We happened to build this image layered on our base image, rather than upon our build_base image. This is because of the order in which we were doing the development for this study. It would probably be best to base the *art* 1.18 image on the build_base image.

```
FROM paterno/centos67base
MAINTAINER XXXXX
LABEL Vendor="FNAL"
LABEL License="BSD"
RUN mkdir /products
RUN cd /products &&
curl -O http://scisoft.fnal.gov/scisoft/bundles/tools/pullProducts &&
chmod u+x pullProducts &&
./pullProducts \$PWD slf6 art-v1_18_03 e9 prof
COPY build_base_setup.sh /etc/build_base_setup.sh
RUN echo 'source /etc/build_base_setup.sh' >> /etc/profile
CMD ["/bin/bash"]
```

Listing 3.3: Dockerfile used to construct the *art* 1.18 image from the base.

Listing 3.4 shows the Dockerfile that was used to construct the study image. This image contains all that is necessary to do development tasks using the *art* framework. The directory `/study` comes pre-populated with the files necessary to begin development work immediately.

```
FROM centos67-art_1_18_03_e9-prof
MAINTAINER XXXXX
LABEL Vendor="FNAL"
LABEL License="BSD"
ADD AnalysisWork/docker_study /study
RUN echo 'cd /study; source setup.sh' >> /etc/profile
CMD ["/bin/bash", "-l"]
```

Listing 3.4: Dockerfile used to construct the study image from *art* 1.18.

Listing 3.5 shows the steps used to build the images, to tag them, and to upload them to a public repository. We used the central docker repository at https://hub.docker.com/, using (Dockerhub) accounts *jbkowalkowski* and *paterno* for pushing images from woof.fnal.gov and the other VMs. In some more recent work, we have begun using *makefiles* to automate these steps.

```
> git clone git@github.com:jbkowalkowski/Dockerfiles.git docker
> cd docker
> cd base_centos67
> docker build -t base_centos67 .
> docker tag -f <IMAGE ID> paterno/centos67base
> docker push paterno/centos67base
> cd ../art_1_18_03
> docker build -t centos67-art_1_18_03_e9-prof .
> docker tag -f <IMAGE ID> jbkowalkowski/centos67-art_1_18_03_e9-prof
> docker push  jbkowalkowski/centos67-art_1_18_03_e9-prof
> cd ../study_1_18_03
> docker build -t centos67-study_1_18_03_e9-prof .
> docker tag -f <IMAGE ID> jbkowalkowski/centos67-study_1_18_03_e9-
   prof
> docker push jbkowalkowski/centos67-study_1_18_03_e9-prof
```

Listing 3.5: Complete set of commands usted to build, tag, and push all the images. The IMAGE ID was obtained using the `docker images` command.

3.3 Using images

For testing on standard Linux machines, the `docker run` command can be used to launch a container. Listing 3.6 provides some example commands. Note the use of `--rm` and `-v` in the command. The flag `-v hostpath:containerpath` instructs Docker to mount the host directory `hostpath` as `containerpath` in the running container. The flag `--rm` instructs Docker to remove the named container upon exit. The combined result is that only changes made to the host filesystem will remain after the container is exited.

```
# Run a container based on the centos67-study_1_18_03_e9-prof image,
# and destroy the container upon exit.
> docker run --rm -it centos67-study_1_18_03_e9-prof
# Mount a directory from the host's scratch filesystem at /scratch in
# the container.
> docker run --rm -it -v /mnt/disk1/scratch/jbk/:/scratch \
    centos67-study_1_18_03_e9-prof
```

Listing 3.6: Examples shell commands to run a Docker container.

Use of Docker images on Cori is different. Cori uses a custom container loading system, developed at NERSC, called Shifter. Shifter is capable of downloading Docker images from Dockerhub. Docker images can be installed from the interactive nodes, but containers can only be run within the compute nodes, launched through the SLURM batch system tools. Shifter provides its own tool, `shifterimg`, to pull Docker images from Dockerhub. Listing 3.7 shows the steps necessary to pull some Docker images.

```
> module load shifter
> shifterimg -v pull docker:paterno/centos-uboone_v04_31_00-e9-prof:
   latest
> shifterimg -v pull docker:jbkowalkowski/centos67-study_1_18_03_e9-
   prof:latest
```

Listing 3.7: The steps necessary to pull Docker images on Cori at NERSC.

Launching a job that uses a Docker container is done using SLURM. Listing 3.8 shows a sample SLURM script that runs the MicroBooNE simulation program.

```
#!/bin/bash -l

#SBATCH --image=docker:paterno/centos-uboone_v04_31_00-e9-prof:latest
#SBATCH --partition=debug
#SBATCH --nodes=1
#SBATCH --time=00:09:30
#  --volume=\$SCRATCH/test\_out:/scratch

shifter --volume=\$SCRATCH/test_out:/mnt /bin/bash -cl "cd /mnt; lar
   -c prodsingle_uboone_mfp_01.fcl >out01.txt 2>&1 "
shifter --volume=\$SCRATCH/test_out:/mnt /bin/bash -cl "cd /mnt; lar
   -c prodsingle_uboone_mfp_02.fcl >out02.txt 2>&1"
```

```
shifter --volume=\$SCRATCH/test_out:/mnt /bin/bash -cl "cd /mnt; lar
    -c prodsingle_uboone_mfp_03.fcl >out03.txt 2>&1 "
```

Listing 3.8: The SLURM batch script for using our Docker container.

4 Results and Discussion

4.1 Results

Two distinct tests were run on each of three different platforms, using the two final images that we produced: one test for the MicroBooNE image and one test for the *art* study image. Table 1 shows the list of platforms used for these two tests, along with a few important attributes.

test	location	model	speed	micro-architecture
cori	cori.nersc.gov	Intel E52698v3	2.30GHz	Haswell
native	woof.fnal.gov	Intel E52680v2	2.80GHz	Ivy Bridge
docker	woof.fnal.gov	Intel E52680v2	2.80GHz	Ivy Bridge

Table 1: Processors used in test

The MicroBooNE test used a standard simulation workflow configuration to provide a set of events; the FHiCL file used to configure the program is shown in listing 4.1.

Listing 4.1: The FHiCL file used to run the MicroBooNE simulation tests.

```
#include "services_microboone.fcl"
#include "singles_microboone.fcl"
#include "largeantmodules_microboone.fcl"
#include "detsimmodules_microboone.fcl"
#include "triggersim_microboone.fcl"
#include "opticaldetectorsim_microboone.fcl"
#include "mccheatermodules.fcl"
process_name: SinglesGen

services: {
  TFileService: { fileName: "single_hist_uboone_01.root" }
  TimeTracker:       { dbOutput: { filename: "timing_01.db" } }
  RandomNumberGenerator: {}
  user:         @local::microboone_full_services
}
services.user.BackTracker: @local::microboone_backtracker

source:{
  module_type: EmptyEvent
  timestampPlugin: { plugin_type: "GeneratedEventTimestamp" }
  maxEvents:   5 firstRun:    1  firstEvent:  1
}
physics:{
 producers:{
   generator:    @local::microboone_singlep
   largeant:     @local::microboone_largeant
   backtrack:    @local::standard_backtrackerloader
   optdigitizer: @local::microboone_optical_adc_sim
   optfem:       @local::microboone_optical_fem_sim
   triggersim:   @local::ubtrigger_singlep
   optreadout:   @local::microboone_optical_dram_readout_sim
   daq:          @local::microboone_simwire
 }
 analyzers:{ largana:   @local::microboone_largeantana }

 simulate: [ generator, largeant, backtrack, optdigitizer, optfem, triggersim,
    optreadout, daq ]
 analyzeIt:  [ largana ]
 stream1:  [ out1 ]
 trigger_paths: [simulate]
 end_paths:     [analyzeIt, stream1]
}
outputs:{
out1:{
   module_type: RootOutput
   fileName:    "single_gen_uboone_01.root"
   compressionLevel: 1
} }

physics.producers.generator.SigmaThetaXZ: [ 5.0 ]
physics.producers.generator.SigmaThetaYZ: [ 5.0 ]
physics.producers.generator.X0: [ 100.0 ]
physics.producers.generator.Z0: [ 50.0 ]
physics.producers.generator.P0: [ 1.5 ]
```

The *art* study test used a test module for producing 500 events, each containing a vector of of 250 thousand integers. Each of the configurations used the `RootOutput` module that writes all the events into an *art*-ROOT file.

Figure 2 on the following page shows the timing for each of the 11 *art* modules used in the test. Note that the distribution of times for the Docker and the native runs on woof are very similar. We see no important performance degradation caused by the user of Docker. The comparison with execution under Docker on Cori shows the same relative weights among the modules. We note that Cori has a significantly slower clock speed than does woof. However, it appears that Cori's Hazwell architecture, which is newer than woofs, makes up for the difference in clock speed to yield very similar timing results.

Figure 3 on page 13 summarizes the timing data by combining the times for all the non I/O modules, to make it easier to compare the compute performance and the I/O performance among the different platforms. The spread of execution times on woof, both for compute modules and for the I/O module, is slightly larger under Docker than on the native run. However, the median times are very similar, and no large outliers are present. We note that both the compute and I/O performance under Docker on Cori a just slightly better than the performance on woof. Table 2 on the following page contains a detailed comparison of the median event processing times.

4.2 Discussion

There are two things that were not addressed in this demonstration: (1) user permission and account access, and (2) shared file systems access through the container.

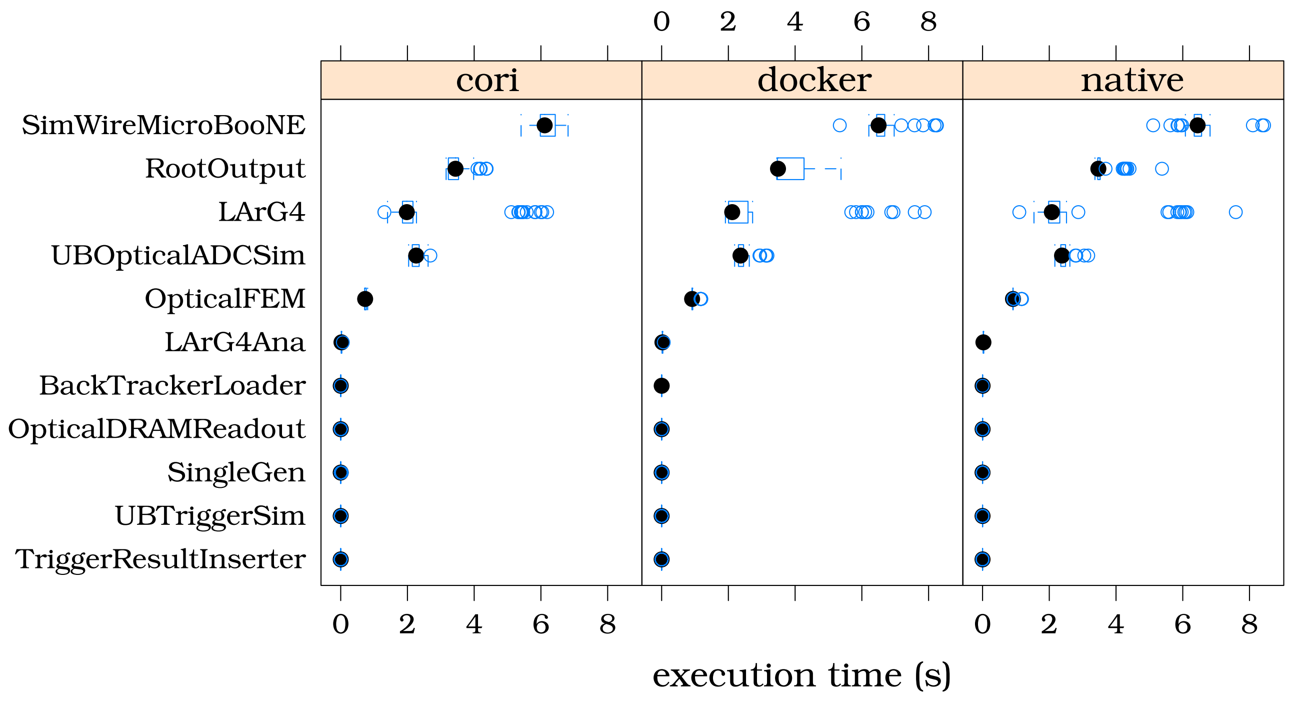

Figure 2: Module timing plot, showing the time in seconds it takes for each module (denoted by the module class name) to process each event, on each platform.

test	modules	mediantime (sec/event)	speed relative to native (larger is faster)
cori	workers	11.325	1.047
docker	workers	11.906	0.996
native	workers	11.853	1.0
cori	output	3.442	1.008
docker	output	3.488	0.995
native	output	3.470	1.0

Table 2: median Per-event timing

The default options for our containers launch the docker image as user *root*. Although the running container is in a closed-off sandbox and damage that can occur appears to be confined, we note that filesystems from the host OS can, and will need to be, mounted within the running container. Access to those filesystems will be allowed according to the accounts within the container. If the container is entered as *root*, then the applications running into the container will be able to write anywhere in the mounted filesystem. NERSC protects the main filesystems from this by booting the container in space that matches the user ID that launched the container. This is an important thing to do. Docker provides facilities to create users within an image, but we have not yet investigated these options.

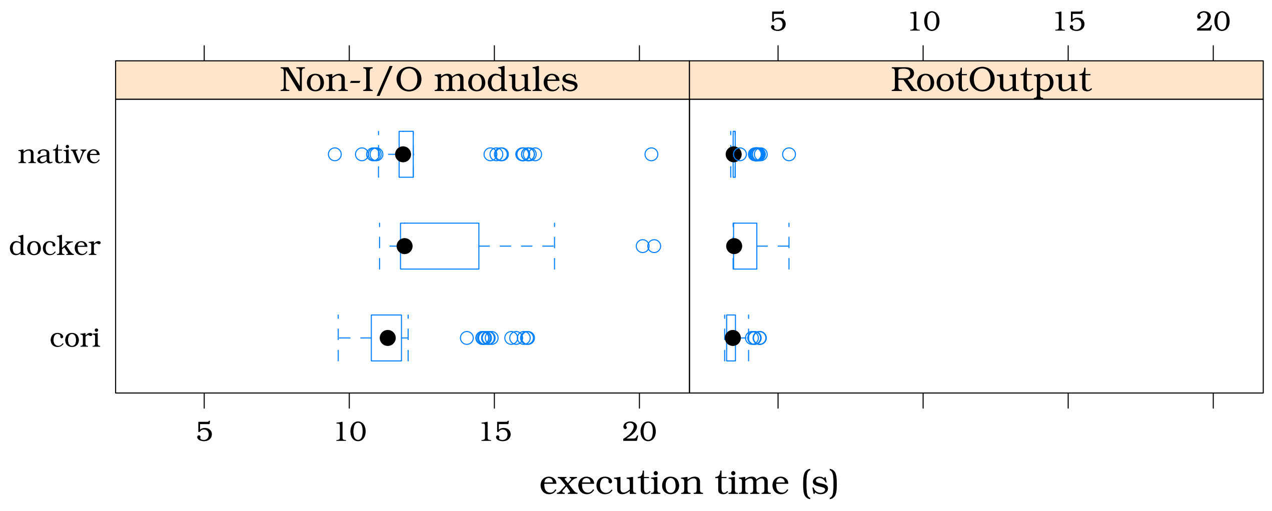

Figure 3: Overall timing plot, showing the summarized per-event timing of the output module (`RootOutput`) compared with the sum of the time taken by all other modules, for each test platform.

In some cases, it is not reasonable to require that the image should contain all applications and data that are necessary for that image to be useful. For data and applications to be reused throughout many runs (or launches) of the container, the shared systems are a good solution. This came up in the context of both of our demonstration containers. For the MicroBooNE container, the timing data and the FHiCL configuration files for starting the jobs were placed into and used out of the Cori global scratch space. For the study container, the situation was more complex. The development space within the container was mounted on `/study` and the Cori scratch space was mounted on `/scratch`. The source code for the test modules was made available in `/study`, along with everything needed to build them. The first job launched completed a build of the test modules into the `/scratch` area. Subsequent jobs directly used the code that was installed into the mounted `/scratch` area. This was feasible because the global scratch area on Cori has a long lifetime and is independent of any one job lifetime.

The entire issue of configuration file, application, and library sharing beyond a basic self-contained software release will need to be addressed. All of these are examples of things that need to be modified and survive outside the scope of a single container launch (batch job in the case of Cori). There are also things that need to be moved in and out of the machine or facility independent of the work carried out inside a container instance.

5 Future directions

This early work demonstrating container technology has exceeded our expectations. All of the tools we've used from the Docker community have worked as promised, and also provide reasonable documentation. This section describes tasks that should be carried out to futher this work.

Providing distributions through container technology looks to be an excellent way to download and run from a complete and preconfigured software distribution. We already need to provide two external groups with a way to confnigure and run our software to produce simulated LAr detector data. Our current plan is to provide these groups with containers to generate the data they need.

5.1 Scaling study with HEP Cloud

MicroBooNE Monte Carlo production is good candidate for Cori. There are a few tasks that need to be completed before demonstrating a production run demonstration on the order of 100,000 cores.

1. Demonstrate configuration and running on 128 Cori cores, using four fully loaded nodes. This will check for CPU and file system performance degradation and also allow us to refine the batch submission scripts. Expand the test to include the hyperthreaded cores (up to 64) to measure the affects on performance.
2. Open a discussion with NERSC on storage and compute cycles for a larger-scale test. This may need to include discussion of WAN bandwidth available from Fermilab to NERSC for moving data off the global file system onto Fermilab resources.
3. Begin discussions with HEP Cloud to demonstrate launching a job through their facilities for running on HPC resources. This will include discussing how to configure individual application instances, how to move produced data back to Fermilab, and any requires we may have for use of CVMFS.

5.2 Image build and deployment

A more automated procedure for building images will be needed for interaction with outside groups. This includes adding steps to the release building procedure to create and publish required images. Further investigate is needed to look into addition features of Docker to aid in the deployment of development images. There are facilities for automating the incorporation of newly built applications and libraries into images that need to be explored. The image layers presented here suited the purposes we had for this demonstration and may need to be reviewed as further tests are carried out on additional platforms.

The last step in our original goal also needs to be completed. This goes a big step further by moving many of the external products directly onto the platform. Such a change permits a customized platform to be built up that meets our specific needs. This is particularly important for core tools such as the C++ compiler, python, and boost.

Bibliography

[1] Docker is described at https://docs.docker.com/engine/introduction/understanding-docker/.

[2] Shifter is described at `https://www.nersc.gov/research-and-development/user-defined-images`.

[3] Dockerhub can be found at `https://hub.docker.com`.

[4] Cori is NERSC's newest supercomputer, and is described at `http://www.nersc.gov/users/computational-systems/cori/cori-phase-i`.

[5] *ISO/IEC 14882:2003 (Second Edition) Programming Languages - C++*, John Wiley & Sons, Ltd.

[6] Initial set of container tasks for testing at `https://cdcvs.fnal.gov/redmine/projects/art-hpc/wiki/Docker_use`.

[7] The BTRFS filesystem is described at https://btrfs.wiki.kernel.org.