Container Computing for HPC and Scientific Workflows - SC16 Tutorial

Container computing is revolutionizing the way applications are developed and delivered. It offers opportunities that never existed before for significantly improving efficiency of scientific workflows and easily moving these workflows from the laptop to the supercomputer. Tools like Docker and Shifter enable a new paradigm for scientific and technical computing. However, to fully unlock its potential, users and administrators need to understand how to utilize these new approaches. This tutorial will introduce attendees to the basics of creating container images, explain best practices, and cover more advanced topics such as creating images to be run on HPC platforms using Shifter. The tutorial will also explain how research scientists can utilize container-based computing to accelerate their research and how these tools can boost the impact of their research by enabling better reproducibility and sharing of their scientific process without compromising security.

The content for the handouts and slides will be posted and updated at https://github.com/luizirber/2016-11-14-sc16.

Prerequisites

This is hands-on tutorial. Participants should bring a laptop and pre-install software in advance to make the best use of time during the tutorial. Participants should install Docker locally on their laptop. Alternatively, participants can have remote access to a system that has docker installed and allows them to run docker commands. Users can also create a dockerhub account in advance at https://hub.docker.com/. This account will be needed to create images on dockerhub. In addition, users should install an ssh client for their operating system so they can access the HPC resources we will use for the Shifter portion of the tutorials. We will attempt to have some alternative options for users who have issues installing Docker.

Agenda

- 8:30: Welcome and Intro to Docker
- 9:00: First hands-on
  - Pulling and running an existing image
  - Making changes and committing them
  - Creating and building a Dockerfile
  - Pushing a Dockerfile to dockerhub
- 9:30: Break (check time on this)
  - Distribute NERSC logins - Please obtain a NERSC login from tutorial staff

- 10:00: Intro to Shifter and how it is different
- 10:30: Second hands-on - Shifter
  - Logging in to NERSC
  - Pulling an image
  - Running an image interactively
  - Submitting a Shifter batch job
  - Running a parallel Python MPI job
- 11:15: Uses Cases in the Real World and Best Practices
  - Reproducibility
  - LHC, Astronomy
- 11:30: Final Hands-on
  - Controlling layers and making builds faster
  - What goes in the image and what should stay out
  - Bring us your problem

Welcome and intro to Docker

Intro to Docker

Pulling and running an existing image

Pull a public image such as ubuntu or centos using the docker pull command. If a tag is not specified, docker will default to "latest".

```
docker pull ubuntu
```

Now run the image using the docker run command. Use the "-it" option to get an interactive terminal during the run.

```
docker run -it ubuntu
```

Making changes and committing them

Using standard linux commands, modify the image.

```
docker run -it ubuntu
echo root@949eb1a6a099:/# (echo '#!/bin/bash'|echo "echo 'Hello World'") > /bin/hello
chmod 755 /bin/hello
# Test it
hello
# Exit
exit
```

Now find the container and commit the changes to a new image called hello.

```
docker ps -a|head -2
# Grab the Container ID
docker commit <ID> hello
```

Now try running the new image with your changes.

```
docker run -it hello
hello
```

Creating and building a Dockerfile

While manually modifying and commiting changes is one way to build images, using a Dockerfile provides a way to build images so that others can understand how the image was constructed and make modifications.

A Dockerfile has many options. We will focus on a few basic ones (FROM, MAINTAINER, ADD, and RUN)

Create a simple shell script called script in your local directory using your favoriate editor.

```
#!/bin/bash
echo "Hello World"
```

Noe create a Dockerfile called Dockerfile in the same directory with contents similar to this. Use your own name and e-mail for the maintainer. Feel free to change the FROM line if there is a different base OS you prefer (e.g. centos or fedora).

```
FROM ubuntu
MAINTAINER Joe Smith <joe@user.com>

ADD . /src
RUN cp /src/script /bin/hello && chmod 755 /bin/hello
```

Now build the image using the docker build command. Be sure to use the -t option to tag it. Tell the Dockerfile to build using the current directory by specifying '.'. Alternatively you could place the Dockerfile and script in an alternate location and specify that directory in the docker build command.

```
docker build -t hello:1.0 .
```

Try running the image.

```
docker run -it hello:1.0
/bin/hello
```

Pushing a Dockerfile to dockerhub

Docker provides a public hub that can be use to store and share images. Before pushing an image, you will need to create an account at Dockerhub. Go to https://hub.docker.com/ to create the account. Once the account is created, push your test image using the docker push command. In this example, we will assume the username is patsmith.

```
docker tag hello:1.0 patsmith/hello:1.0
docker push patsmith/hello:1.0
```

The first push make take some time depending on your network connection and the size of the image.

Intro to Shifter and how it is different

A brief presentation on Shifter will be given. This will include an explanation of how Shifter works and how the impacts the way images should be prepared.

Second hands-on - Shifter

Logging in to NERSC

Use ssh to connect to Cori. The username and password will be on the training account sheet.

```
ssh <account>@cori.nersc.gov
```

Pulling an image

Pull an image using shifterimg. You can pull a standard image such as Ubuntu or an image you pushed to dockerhub in the previous session.

```
shifterimg pull ubuntu:14.04
# OR
shifterimg pull scanon/shanetest:latest
```

Running an image interactively

Use salloc and shifter to test the image.

```
salloc -N 1 --image ubuntu:14.04
shifter bash
```

You should be able to browse inside the image and confirm that it matches what you pushed to dockerhub earlier.

```
ls -l /app
lsb_release -a
```

Once you are done exploring, exit out.

```
exit
exit
```

Submitting a Shifter batch job

Now create a batch submission script and try running a batch job with shifter. Use vi or your other favorite editor to create the submission script or cat the contents into a file.

```
cat << EOF > submit.sl
#!/bin/bash
#SBATCH -N 1
#SBATCH --image ubuntu:latest

srun -N 1 shifter /app/app.py
EOF
```

Use the Slurm sbatch command to submit the script.

```
sbatch ./submit.sl
```

Running a parallel Python MPI job

It is possible to run MPI jobs in Shifter and obtain native performance. There are several ways to achieve this. We will demonstrate one approach here.

On your laptop create and push a docker image with MPICH and a sample application installed.

First, create and save a Hello World MPI application.

```
// Hello World MPI app
#include <mpi.h>

int main(int argc, char** argv) {
    int size, rank;
    char buffer[1024];

    MPI_Init(&argc, &argv);

    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    gethostname(buffer, 1024);

    printf("hello from %d of %d on %s\n", rank, size, buffer);
    MPI_Barrier(MPI_COMM_WORLD);

    MPI_Finalize();
    return 0;
}
```

Next create, a Dockerfile that install MPICH and the application.

```
# MPI Dockerfile
FROM ubuntu:14.04
RUN apt-get update && \
    apt-get install -y autoconf automake gcc g++ make gfortran
ADD http://www.mpich.org/static/downloads/3.2/mpich-3.2.tar.gz /usr/local/src/
RUN cd /usr/local/src/ && \
    tar xf mpich-3.2.tar.gz && \
    cd mpich-3.2 && \
    ./configure && \
    make && make install && \
    cd /usr/local/src && \
    rm -rf mpich-3.2

ADD helloworld.c /
RUN mkdir /app && mpicc helloworld.c -o /app/hello

ENV PATH=/usr/bin:/bin:/app
```

Build and push the image.

```
docker build -t <mydockerid>/hello:latest .
docker push <mydockerid>/hello:latest
```

Next, return to your Cori login, pull your image down and run it.

```
shifterimg pull <mydockerid>/hello:latest
#Wait for it to complete
salloc -N 2 --image <mydockerid>/hello:latest
# Wait for prepare_compilation_report
# Cori has 32 physical cores per node with 2 hyper-threads per core.
# So you can run up to 64 tasks per node.
srun -N 2 -n 128 shifter /app/hello
exit
```

If you have your own MPI applications, you can attempt to Docker-ize them using the steps above and run it on Cori. As a courtesy, limit your job sizes to leave sufficient resources for other participants. *Don't forget to exit from any "salloc" shells once you are done testing.*

Uses Cases in the Real World and Best Practices

We will present some examples and best practices from real world use cases. This will
include insites on how Docker can be used to assist with reproducibility.

Containers and Reproducibility

Use Cases: Large Hadron Collider and Astronomy

Final Hands-on

Controlling layers and making builds faster

How you construct your Dockerfile can have a big impact on your image sizes. Keeping images compact, decreases the time to pull an image down or convert images for use by Shifter. Here are a few tips to reduce image sizes.

Cleanup within the construction of a layer

Each RUN statement will result in a new layer. For example, let's look at the following.

```
RUN wget http://hostname.com/mycode.tgz
RUN tar xzf mycode.tgz
RUN cd mycode ; make; make install
RUN rm -rf mycode.tgz mycode
```

This will result in four layers. Unfortunately, the cleanup line in the last RUN line will not reduce the amount of data that must be pulled from the Docker registry. Instead, it will mask or "white-out" the files so they don't appear when executing the image.

In contrast, let's examine the following.

```
RUN wget http://hostname.com/mycode.tgz && \  tar xzf mycode.tgz &&
\  cd mycode && make && make install && \  rm -rf mycode.tgz mycode
```

This image size will be much smaller since the cleanup happens inside the layer construction. Also, notice the use of "&&" between commands. This is good practice so that failures are detected and will stop the build. Using a semi-colon like was done above means that the make could fail but the build would continue on.

What goes in the image and what should stay out

While Docker doesn't typically impose strict limits on image sizes, larger images are more prone to failure for a variety of reasons. Some Docker deployments may have limited disk space to store images. Pulling down large layers can trigger timeouts or other failures. Here are some best practices to follow.

- Limit image sizes to a few GB. Avoid exceeding 10GB or more.
- Avoid "Kitchen sink" images that contain extraneous applications or tools
- Limit layer sizes using the technique above
- Avoid including data sets in the image unless they are relatively small and static. Data can be mapped into the image using volume mounts (-v option).
- Limit images to the specific target applications and only add what is needed to support that application.

Bring us your problem

If you have completed the exercises and there is time remaining. We encourage you to try to Docker-ize a real application. Please, ask the tutorial staff for help.