

Toward using your clunky software anywhere: Computing with containers

Andrew Monaghan

Andrew.Monaghan@colorado.edu

Daniel Trahan

Daniel.Trahan@colorado.edu

Slides and exercises available for download at:

https://github.com/rctraining/crdds_2018_container_tutorial/

Outline

Part 1: Container fundamentals and Docker

- Introduction to containers
- Docker commands and options
- Hands-on: Running Docker containers on your personal Machine
- Hands-on: Building Docker images
- Hands-on: Practical Application

Part 2: Containers for HPC with Singularity

- Singularity commands and options
- Hands-on: Running containers on RMACC Summit
- Building containers
- Special cases: Running containers for MPI and GPU jobs

Introduction to Containers



What is a container?

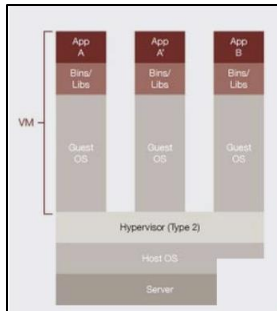
A container is a portable environment that packages some or all of the following: an operating system, software, libraries, compilers, data and workflows. Containers enable:

- Mobility of Compute
- Reproducibility (software and data)
- User Freedom

Virtualization (1)

Hardware virtualization (not used by containers!)

- Can run many OS's on same hardware (machine)
- E.g., VirtualBox, VMWare



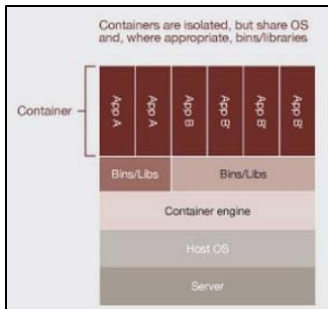
Material courtesy: M. Cuma, U. Utah

Virtualization (2)

OS-level virtualization (used by containers!)



- Can run many isolated OS instances (guests) under a server OS (host)
- Also called containers
- E.g., Docker, Singularity

Best of both worlds: isolated environment that user wants, but can leverage host OS resources (network, I/O partitions, etc.)



Material courtesy: M. Cuma, U. Utah

Containerization software

- Docker 
 - Part 1 focus of today's tutorial
 - Well established – largest user base
 - Has Docker Hub for container sharing
 - Problematic with HPC
- Singularity 
 - Part 2 focus of today's tutorial
 - Designed for HPC
- Charliecloud; Shifter
 - Designed for HPC
 - Based on Docker
 - Less user-friendly

Installing Docker

- Docker Community Edition
 - Windows: Windows 10 Professional or Enterprise
 - Mac: OS X El Capitan 10.11 or later
 - Linux
- Docker toolbox
 - Legacy solution for Windows and Mac for versions that do not meet the version requirements.
 - Utilizes the Virtual Box hypervisor for virtualization
 - For this tutorial, Windows users regardless of version will use Docker toolbox

Why Docker?

- Probably the most popular containerization software
- Offers a variety of prebuilt images including:
 - Python
 - Perl
 - NodeJS
 - Ubuntu
- Very well documented with a large community creating and supporting docker images.
- DockerHub

Docker Nuts and Bolts

- Docker runs on a concept of images and containers.
 - **Images**: Saved snapshots of a container environment.
 - Made from Dockerfile or pulled from Docker Hub
 - Stored in the Docker cache on your disk
 - Immutable
 - **Containers**: Instances of images that are generated by Docker when an image is 'run'
 - Instance of image running in memory
 - Ephemeral and state cannot be saved
 - Can be run interactively

Docker Commands

- Docker Commands are usually in the form of:
`docker <sub-command> <flags> <target/command>`
- Examples:
`docker run -it myimage`
`docker container ls`
`docker image prune`

Running Docker Containers

- Run a docker image as a container:

```
docker run <image-name>
```

- Run a docker image interactively:

```
docker run -it <image-name>
```

- If an image is not on the system, then Docker will search Dockerhub to see if the image exists.
- Specify commands after your image to execute specific software in your container.

```
docker run <image-name> <program>
```

Containerized Hello World

- Let's start with something simple:
 - Docker "Hello, World!"
 - Relatively small image
 - No dependencies
 - Built as a general test case
- Command:

```
docker run hello-world
```

Docker Image and Container Commands

Image Commands	
<code>docker image ls</code>	List docker images stored in cache:
<code>docker image rm <image></code> <code>docker rmi <image></code>	Remove (an) image(s):
<code>docker image prune</code>	Remove unused images
Container Commands	
<code>docker container ls</code>	List docker containers currently running:
<code>docker container rm <container></code> <code>docker rm <container></code>	Remove (an) container(s):
<code>docker container prune</code>	Remove all stopped containers

Docker Utility Commands

Commands	
<code>docker info</code>	Show s Docker system-wide information
<code>docker inspect <docker-object></code>	Show s low -level information about an object
<code>docker config <sub-command></code>	Manage docker configurations
<code>docker stats <container></code>	Show s container resource usage
<code>docker top <container></code>	Show s running processes of a container
<code>docker version</code>	Show s docker version information

- More details and commands can be found [on the docker documentation page](#)

Demo 1: Running Containerized Python

Demo 1: Python

- Running the Python docker container will pull Python from Docker Hub:

```
docker run python:3.7-slim
```

- ...did it work?
- Run your python image interactively:

```
docker run -it python:3.7-slim
```

- This puts us into a python interpreter, where you can run python code containerized in its own environment.

Building Docker Containers

- To build a docker container, we need a set of instructions Docker can use to set up the environment.
 - Dockerfile
- Once we set up our dockerfile we can use the command
`docker build -t <image-name> .`
- Then we can run the image with our `docker run` command
`docker run <image-name>`

Demo 2: Building an Ubuntu Container

Demo 2: Setup

Dockerfiles and test files are provided for this workshop. We can pull the files from a github repository as such:

1. Navigate to your home directory

```
cd ~
```

2. Clone the repository

```
git clone https://github.com/rctraining/crdds_2018_container_tutorial
```

Demo 2: Ubuntu w/ GCC

- For this first example we will be building a custom Ubuntu image that will provide a location to run the GNU Compiler Collection.
- Dockerfile provided
- Need to build:
 1. Navigate to the directory:
`cd ~/crdds_2018_container_tutorial/dockerdemo/ubuntu-gcc`
 2. Build the docker image with:
`docker build -t happy-gcc`
 3. Run the docker image as a container:
`docker run -it happy-gcc`

Mounting and Accessing files

- So now that we have a working container, how can we access the test files we downloaded?
 - Mounting directories: Bind Mount
 - Allows the docker container to access files on the host OS
 - Choose host's **source directory**, files in the directory will be moved to the container's **target directory**
 - **Source Directory**: Directory on the host system.
Never within a container.
 - **Target Directory**: Directory in the Docker Container.
Never on the host system.
 - A flag set within the **docker run** command:

```
docker run --mount type=bind,source=<source>,target=<target> <image>
```

Mounting and Accessing files

- Mounting directories: Volume Mount
 - Same concept, but volumes are stored within docker cache.
 - Create Docker volumes in your terminal and link your volume directory
 - Similarly linked through the `docker run` command.

```
docker run --mount type=volume,source=<volume>,target=<target> <image>
```

Demo 2 (Cont.): Mounting

- Returning to our demo, can we give our container access to our test files?
- Let's use a bind mount!
- In the directory where our Dockerfile lives... use this command (all on one line):

```
docker run -it --source  
type=bind,source=$(pwd)/source,target=/target happy-gcc
```

- We can `cd /target` and run our test files!
- Command:

```
gcc test.c -o test.exe  
./test.exe
```


Demo 3: NCL container

- For this next example we will building a Docker image that will run the NCAR Command Language (NCL).
- Dockerfile provided
- Same process as before:
 1. Navigate to the Dockerfile found at:
[~/crdds_2018_container_tutorial/dockerdemo/ncl](#)
 2. Build your docker file as an image titled "bright-ncl"
 3. Run your docker image as a container
- Can we test a sample script?

Questions?

- Contact: daniel.trahan@colorado.edu

If you will not be attending the 2nd half of the lecture, please fill out the survey:

<http://tinyurl.com/curc-survey18>

Up Next... Singularity



Part 2: Containers for HPC with Singularity



Part 2 Outline

- Singularity commands and options
- Hands-on: Running containers on RMACC Summit
- Building containers
- Special cases: Running containers for MPI and GPU jobs

Why Singularity?

- Singularity is a comparably safe container solution for HPC
 - User is same inside/outside container
 - User cannot escalate permissions without administrative privilege
- Can support MPI and GPU resources on HPC (scaling)
- Can use HPC filesystems
- Supports the use of Docker containers
- Container is seen as a file, and can be operated on as a file

Singularity commands and options



The Singularity workflow

Interactive Development

```
sudo singularity build --sandbox tmpdir/ Singularity
```

```
sudo singularity build --writable container.img Singularity
```

BUILD ENVIRONMENT

Build from Recipe

```
sudo singularity build container.img Singularity
```

Build from Singularity

```
sudo singularity build container.img shub://vsoch/hello-world
```

Build from Docker

```
sudo singularity build container.img docker://ubuntu
```

Container Execution

```
singularity run container.img  
singularity shell container.img  
singularity exec container.img ...
```

Reproducible Sharing

```
singularity pull shub://...  
singularity pull docker://... *
```

PRODUCTION ENVIRONMENT

* Docker construction from layers not guaranteed to replicate between pulls

We will cover this last

We will cover this first

<https://singularity.lbl.gov>

Singularity Commands

build: Build a container on your user endpoint or build environment

exec: Execute a command to your container

inspect: See labels, run and test scripts, and environment variables

pull: pull an image from Docker or Singularity Hub

run: Run your image as an executable

shell: Shell into your image

<https://singularity.lbl.gov>

Running containers

Now, on **any system** with Singularity, even without administrative privilege, you can retrieve and use containers:

- Download a container from Singularity Hub or Docker Hub
 - `singularity pull shub://some_image`
 - `singularity pull docker://some_image`
- Run a container
 - `singularity run mycont.img`
- Execute a specific program within a container
 - `singularity exec mycont.img python myscript.py`
- “Shell” into a container to use or look around
 - `singularity shell mycont.img`
- Inspect an image
 - `singularity inspect --runscript mycont.img`

Hands-on Examples

- Running containers with Singularity



Logging in

If you are using a temporary account: In a terminal or Git Bash window, type the following:

```
ssh userNNNN@tlogin1.rc.colorado.edu
```

Password:

***Note that `userNNNN` is a temporary account that I have assigned to you. If you don't have one, let me know.*

If you are using your regular RC account (e.g., 'monaghaa' for me), log in as you normally would.

Getting on a compute node

Navigate to scompile

```
ssh scompile
```

Now start an interactive job:

```
sinteractive -ntasks=1 --reservation=container1
```

And load singularity:

```
module load singularity
```

And go to your working directory

```
cd /scratch/summit/$USER
```

...We will go through the example together in the following slides.

Running a container from Singularity Hub

Pull an existing container image that someone else posted:

```
singularity pull --name hello.img shub://monaghaa/singularity_git_tutorial
```

...And run it

```
singularity run hello.img
```

...And look at the script inside

```
singularity exec hello.img cat /code/hello.sh
```

Running a container from Docker Hub

Let's grab the stock docker python container:

```
singularity pull --name pythond.img docker://python
```

...And run python

```
singularity exec pythond.img python
```

...And shell into the container and look around

```
singularity shell pythond.img
```

...try `ls /` What directories do you see?

Running a container from Docker Hub (2)

Let's run an external python script using the containerized version of python:

First create a script called *“myscript.py”* as follows:

```
echo 'print("hello world from the outside")' >myscript.py
```

...And now let's run the script using the containerized python

```
singularity exec pythond.img python ./myscript.py
```

...Conclusion: Scripts and data can be kept inside or outside the container. In some instances (e.g., large datasets or scripts that will change frequently) it is easier to containerize the software and keep everything else outside.

Binding directories

On Summit, most host directories are “bound” (mounted) by default. But on other systems, or in some instances on Summit, you may want to access a directory that is not already mounted. Let’s try it.

Note that the “/opt” directory in “pythond.img” is empty. But the Summit “/opt” directory is not. Let’s bind it:

```
singularity shell -bind /opt:/opt pythond.img
```

...It isn’t necessary to bind like-named directories like we did above. Try binding your /home/\$USER directory to /opt.

***Note: If your host system does not allow binding, you will need to create the host directories you want mounted when you build the container (as root on, e.g., your laptop)

Building containers



There are 2 basic ways to build a container

1. Build a container on a system on which you have administrative privilege (e.g., your laptop).
 - **Pros:** You can interactively develop the container.
 - **Cons:** Requires many GB of disk space, requires administrative privilege, must keep software up-to-date, container transfer speeds can be slow depending on personal network connection.
2. Build a container on Singularity Hub
 - **Pros:** Essentially zero disk space required on your system, doesn't require administrative privilege, no software upgrades needed, easy to retrieve from anywhere, typically faster transfers from Singularity Hub to desired endpoint.
 - **Cons:** Cannot interactively develop the container

Container Formats

- **squashfs**: the default container format is a compressed read-only file system that is widely used for things like live CDs/USBs and cell phone OS's
- **ext3**: (also called **writable**) a writable image file containing an ext3 file system that was the default container format prior to Singularity version 2.4
- **directory**: (also called **sandbox**) standard Unix directory containing a root container image
- **tar.gz**: zlib compressed tar archive
- **tar.bz2**: bzip2 compressed tar archive
- **tar**: uncompressed tar archive

<https://singularity.lbl.gov>

1. Building a container on your machine (interactive):

Within your build environment (a laptop, workstation, or a server that you control): develop and test **writable** containers in ext3 format:

- Build from an existing recipe
 - `sudo singularity build --writable tmp.img mycont.def`
- Build from an existing container on Singularity Hub
 - `sudo singularity build --writeable mycont.img shub://somewhere`
- Build from an existing container on Docker Hub
 - `sudo singularity build --writable mycont.img docker://somewhere`
- Can also build a writable directory (“sandbox”) instead of image, e.g.:
 - `sudo singularity build --sandbox tmpdir/ mycont.def`

What is a recipe?

Header

```
Bootstrap:docker  
From:ubuntu:latest
```

Metadata

```
%labels  
MAINTAINER Andy M
```

Runtime
environment
variables

```
%environment  
HELLO_BASE=/code  
export HELLO_BASE
```

Default program
at runtime

```
%runscript  
echo "This is run when you run the image!"  
exec /bin/bash /code/hello.sh "$@"
```

Where software
and directories
are installed at
buildtime

```
%post  
echo "This section is performed after you bootstrap to build the image."  
mkdir -p /code  
apt-get install -y vim  
echo "echo Hello World" >> /code/hello.sh  
chmod u+x /code/hello.sh
```

See <http://singularity.lbl.gov/docs-recipes> for more.

Basic process of building a container interactively

- Bootstrap a base container (has OS you want, maybe other stuff too)
- Shell into the container
 - Install additional needed programs
 - If they have dependencies, install the dependencies – google for the OS provided packages first and install with apt-get/yum if possible
 - Put the commands in the %post scriptlet of the recipe.
- Build the container again
 - Now with the additional commands in the recipe
 - If something fails, fix it, build container again
- Iterate until all needed programs are installed

When you are done building interactively...

After interactive development, within your build environment (a laptop, workstation, or a server that you control): build your production containers with a *squashfs* filesystem...

- Build from a “recipe” that you developed.
 - `sudo singularity build mycont.img mycont.def`
- Build from an existing container on Singularity Hub
 - `sudo singularity build mycont.img shub://somewhere`
- Build from an existing container on Docker Hub
 - `sudo singularity build mycont.img docker://somewhere`

2. Building a container on Singularity Hub (basic steps)

1. Create a recipe file for your container
2. Name it "Singularity"
3. Create a github repository for your container
4. Upload it to your github repository
5. Log into Singularity Hub using your github username/password
6. Go to "My Collections" and choose "ADD A COLLECTION"
7. Select the github repository you just uploaded.
8. The container will build automatically.
9. Revised recipes are automatically rebuilt when pushed to github.
10. Additional details at:

<https://github.com/singularityhub/singularityhub.github.io/wiki>

Special Cases: MPI and GPUs



Running a multi-core (MPI) job with a container

Syntax (after loading singularity and openmpi):

```
"mpirun -np X singularity exec container.img executable-in-container"
```

1. mpirun is called by the resource manager or the user directly from a shell
2. Open MPI then calls the process management daemon (ORTED)
3. The ORTED process launches the Singularity container requested by the mpirun command
4. Singularity builds the container and namespace environment
5. Singularity then launches the MPI application within the container
6. The MPI application launches and loads the Open MPI libraries
7. The Open MPI libraries connect back to the ORTED process via the Process Management Interface (PMI)
8. At this point the processes within the container run as they would normally directly on the host.

<http://singularity.lbl.gov/docs-hpc>

Running containers on GPUs

Syntax (after loading Singularity on a gpu node ["--partition=sgpu"]):

```
singularity exec --nv docker://tensorflow/tensorflow:latest-gpu  
python mytensorflow_script.py
```

With the **--nv** option the driver libraries do not have to be installed in the container. Instead, they are located on the host system (e.g., RMACC Summit) and then bind mounted into the container at runtime. This means you can run your container on a host with one version of the NVIDIA driver, and then move the same container to another host with a different version of the NVIDIA driver and both will work. (Assuming the CUDA version installed in your container is compatible with both drivers.)

The present NVIDIA Driver on RMACC Summit is 390.48, which is compatible with CUDA 9.1 or lower.

This means you can build a container by bootstrapping a base NVIDIA CUDA image (9.1 or lower) from: <https://hub.docker.com/r/nvidia/cuda/>

Source and additional context: <http://singularity.lbl.gov/docs-exec#a-gpu-example>

Thank you!

Please fill out the survey:

<http://tinyurl.com/curc-survey18>

Contact information:

Andrew.Monaghan@Colorado.edu; Daniel.Trahan@Colorado.edu

Additional learning resources:

Slides and Examples from this course:

<https://github.com/rctraining/crdds> *2018 container tutorial*

Web resources:

<https://training.play-with-docker.com> (*docker online training materials*)

<https://singularity.lbl.gov/user-guide> (*user guide for Singularity*)

<https://www.singularity-hub.org/> (*Singularity Hub*)

Extra slides



Troubleshooting (1)

Container/host environment conflicts

- Container problems are often linked with how the container “sees” the host system. Common issues:
 - The container doesn't have a bind point to a directory you need to read from / write to
 - The container will “see” python libraries installed in your home directory (and perhaps the same is true for R and other packages. If this happens, set the PYTHONPATH environment variable in your job script so that it points to the container paths first.
 - `export PYTHONPATH=<path-to-container-libs>:$PYTHONPATH`
- To diagnose the issues noted above, as well as others, “shelling in” to the container is a great way to see what's going on inside. Also, look in the singularity.conf file for system settings (can't modify).

Troubleshooting (2)

Failures during container pulls that are attributed (in the error messages) to `*tar.gz` files are often due to corrupt `tar.gz` files that are downloaded while the image is being built from layers. Removing the offending `tar.gz` file will often solve the problem.

When building ubuntu containers, failures during `%post` stage of container builds from a recipe file can often be remedied by starting the `%post` section with the command `apt-get update`. As a best practice, make sure you insert this line at the beginning of the `%post` section in all recipe files for ubuntu containers.

<https://singularity.lbl.gov>

Caveats (1)

We didn't cover overlays. These are additional images that are "laid" on top of existing images, enabling the user to modify a container environment without modifying the actual container. Useful because:

1. Overlay images enable users to modify a container environment even if they don't have root access (though changes disappear after session)
2. Root users can permanently modify overlay images without modifying the underlying image.
3. Overlays are a likely way to customize images for different HPC environments without changing the underlying images.
4. More on overlays: <http://singularity.lbl.gov/docs-overlay>

We didn't cover GPU usage with containers. See:
<http://singularity.lbl.gov/archive/docs/v2-3/tutorial-gpu-drivers-open-mpi-mtls>

Caveats (2): Moving containers

You've built your first container on your laptop. It is 3 Gigabytes. Now you want to move it to Summit to take advantage of the HPC resources. What's the best way?

Remember, containers are files, so you can transfer them to Summit just as you would a file:

- Command line utilities (scp, sftp)
- **Globus** (recommended)
- For more on data transfers to/from Summit:
<https://github.com/ResearchComputing/Research-Computing-User-Tutorials/wiki/Data-Transfers>

Running a multi-core (MPI) container: challenges (1)

What are supported Open MPI Version(s)? To achieve proper containerized Open MPI support, you should use Open MPI version 2.1. There are however three caveats:

Open MPI 1.10.x may work but we expect you will need exactly matching version of PMIx and Open MPI on both host and container (the 2.1 series should relax this requirement)

Open MPI 2.1.0 has a bug affecting compilation of libraries for some interfaces (particularly Mellanox interfaces using libmxm are known to fail). If you are in this situation you should use the master branch of Open MPI rather than the release.

Using Open MPI 2.1 does not magically allow your container to connect to networking fabric libraries in the host. If your cluster has, for example, an infiniband network you still need to install OFED libraries into the container. Alternatively you could bind mount both Open MPI and networking libraries into the container, but this could run afoul of glibc compatibility issues (its generally OK if the container glibc is more recent than the host, but not the other way around)

Source (and additional context): <http://singularity.lbl.gov/docs-hpc>

Running a multi-core (MPI) container: challenges (2)

As alluded to in the previous slide, most Singularity examples and documentation are based on OpenMPI. In turn, OpenMPI is somewhat unique because it is not application binary interface (ABI) compatible with other MPI distributions such as MPICH, MVAPICH2 and Intel MPI.

This lack of compatibility contributes to the issues noted in the previous slide, in which a container with a given OpenMPI build may not work properly on hosts with other MPI distributions or different versions of OpenMPI.

MPICH has a greater degree of ABI compatibility. Therefore, at present it may be easier to build containers with parallel codes using MPICH (or Intel MPI) rather than OpenMPI.

If you have questions, contact one of us (our email addresses are on the title slide).

Alternative to Singularity on HPC systems: Udocker

Python-based tool for executing Docker containers without requiring administrative privilege (i.e., can be used on RMACC Summit).

Alternative to using Singularity for Docker containers.

User can install in seconds without needing root access.

Main advantages: Provides more flexibility to modify Docker containers on RMACC Summit (e.g., user can install software in existing container as root, and can install python/R packages).

Main disadvantages: Does not work for Singularity containers; Cannot create containers from scratch.

https://github.com/indigo-dc/udocker/blob/master/doc/user_manual.md

Easiest way to teach is by example....

On your personal machine: How to start your Linux VM so that you can run Singularity

Once you've installed Singularity on your machine (assuming Mac or Windows):

```
$ mkdir singularity-vm  
$ cd singularity-vm  
$ vagrant init singularityware/singularity-2.5.2  
$ vagrant up  
$ vagrant ssh
```

If you are on Linux, you don't need to start a VM