Containers@TACC Documentation

Release 0.1

Texas Advanced Computing Center

Nov 06, 2020

Contents:

Software containers are an important common currency for portable and reproducible computing. Learn best practices on building, using, and sharing Docker and Singularity containers in this hands-on workshop. Also learn how to run those containers on TACC HPC systems, including MPI and GPU aware containers.

Topics will include:

- Docker and Singularity basics
- Containerizing your own code
- Running containers at TACC, including MPI parallelism and GPU enabled containers
- Integration with BioContainers and the module system

CHAPTER 1

Welcome to Containers @ TACC!

In this section, we will learn about containers, their uses, different existing container technologies. Our focus would be on one such container technology called "Docker".

Objectives for this session

- To understand Containers
- To find and use existing containers
- To develop, use your own containers

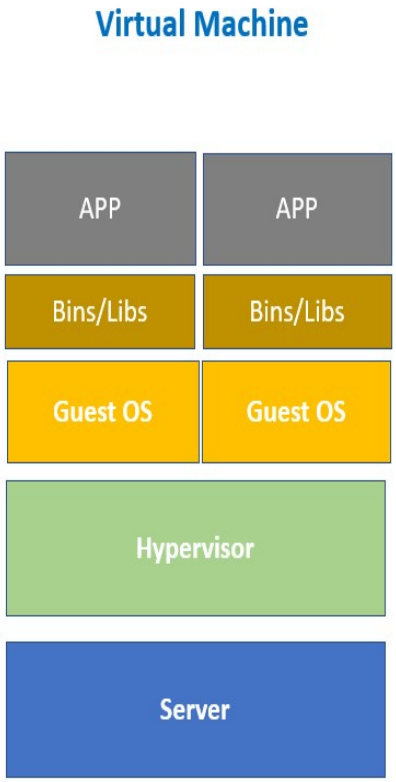
1.1 Introduction to Containers

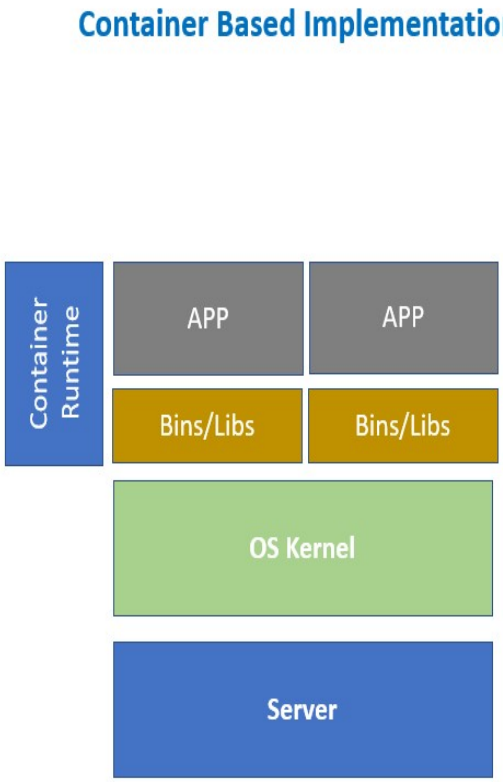
1.1.1 What is a Container?

- A container is a standard unit of software that packages up code and all its dependencies so the application runs quickly and reliably from one computing environment to another.
- Containers allow a developer to package up an application with all of the parts it needs, such as libraries and other dependencies, and ship it all out as one package.
- Multiple containers can run on the same machine and share the OS kernel with other containers, each running as isolated processes in user space, hence are *lightweight* and have *low overhead*.
- Containers ensure *portability* and *reproducibilty* by isolating the application from environment.

Container vs VM

Container	VM
OS process level isolation	OS level isolation with virtualized hardware
Can run 1,000s on a single machine	Can run "dozens" on a single machine.
Leverages kernel features (requirements on kernel version)	Leverages hypervisors (requirements on hardware)
Start up time ~100s of ms	Start up time ~minutes


Virtual Machine
APP
APP
Bins/Libs
Bins/Libs
Guest OS
Guest OS
Hypervisor
Server


Container Based Implementation
Container Runtime
APP
APP
Bins/Libs
Bins/Libs
OS Kernel
Server

Benefits of using containers include:

- Platform independence: Build it once, run it anywhere
- Resource efficiency and density
- Enables reproducible science
- Effective isolation and resource sharing

Container Technologies

Docker

Docker is a set of platform as a service products that use OS-level virtualization to deliver software in packages called containers.

Singularity

Singularity is a container solution which provides the functionality of a virtual machine, without the heavyweight implementation and performance costs of emulation and redundancy. Software developers can now build their stack onto whatever operating system base fits their needs best, and create distributable runtime encapsulated environments and the users never have to worry about dependencies, requirements, or anything else from the user space.

Docker helps to develop containers and run them on our laptops. We use Singularity as a runtime on our HPC systems.

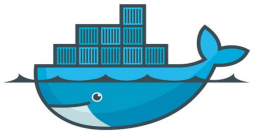
We can find existing containers at:

1. Docker Hub
2. Singularity Hub
3. BioContainers

Docker

Docker is a platform for building and executing containers.

- Docker (2013) is the gold standard container technology.
- It can package an application and its dependencies in a virtual container that can run on any Linux server.
- This helps provide **flexibility** and **portability** enabling the application to be run in various locations.
- Docker grants superuser privileges and some containers may allow users root access to host files.
- Docker-compatible technologies Singularity(Stampede2) and Shifter (Blue Waters, Cori) were designed for HPC environments.



In the Docker world . . .

Containers

A container is a standard unit of software that packages up code and all its dependencies so the application runs quickly and reliably from one computing environment to another. Containers includes everything from the operating system, user-added files, metadata.

Containers are very useful as they are available for both Linux and Windows-based applications, containerized software will run the same, irrespective of the infrastructure. They isolate the software from the environment, facilitating users to overcome conflicts between local development environments and execution environment.

Running an image creates a Docker Container They feature a read-write layer on top, are interactive and can store state. This means that once you execute inside a Docker container, you can save a snapshot of the resulting state as another image.

Images

A Docker image is a read-only file used to produce Docker containers. It is comprised of layers of other images, any change made to an image is carried out by adding a new layer.

Base Image is the layer that does not depend on any other layer. For example most of the time this layer defines the operating system for the docker container's environment.

An image gets built by building a Dockerfile.

. . . which brings us to

What is a Dockerfile?

Dockerfiles are text files you create with the commands you would like to execute on the command line inside a container to create a Docker image. Docker reads the commands from top to bottom and builds a Docker image from it.

Dockerfiles are useful as they contain the history of the procedure used to create an image. They can be used to install required dependencies, tools, tool-related files into the docker container.

Some best practices for writing Dockerfiles can be found at: Best Practices.

The *docker build* command builds an image from a Dockerfile and a context. The build's context is the set of files at a specified location PATH or URL. The PATH is a directory on your local filesystem. The URL is a Git repository location.

With a Dockerfile in the current directory, we can build an image from it by

```
$ docker build .
Sending build context to Docker daemon  6.51 MB
. . .
```

Image Registry

We can store the docker images we create in image registries. Registries are organized into collections of images called *repositories*.

Docker Hub is a central, public repository of images. The docker hub contains images contributed by individual users and organizations as well as "official images". Explore the offcial docker images here: https://hub.docker.com/explore/

Image Tags

Docker supports the notion of image tags, similar to tags in a git repository. Tags identify a specific version of an image.

The full name of an image on the Docker Hub is comprised of components separated by slashes. The components include a "repository" (which could be owned by an individual or organization), the "name", and the "tag". For example, an image with the full name

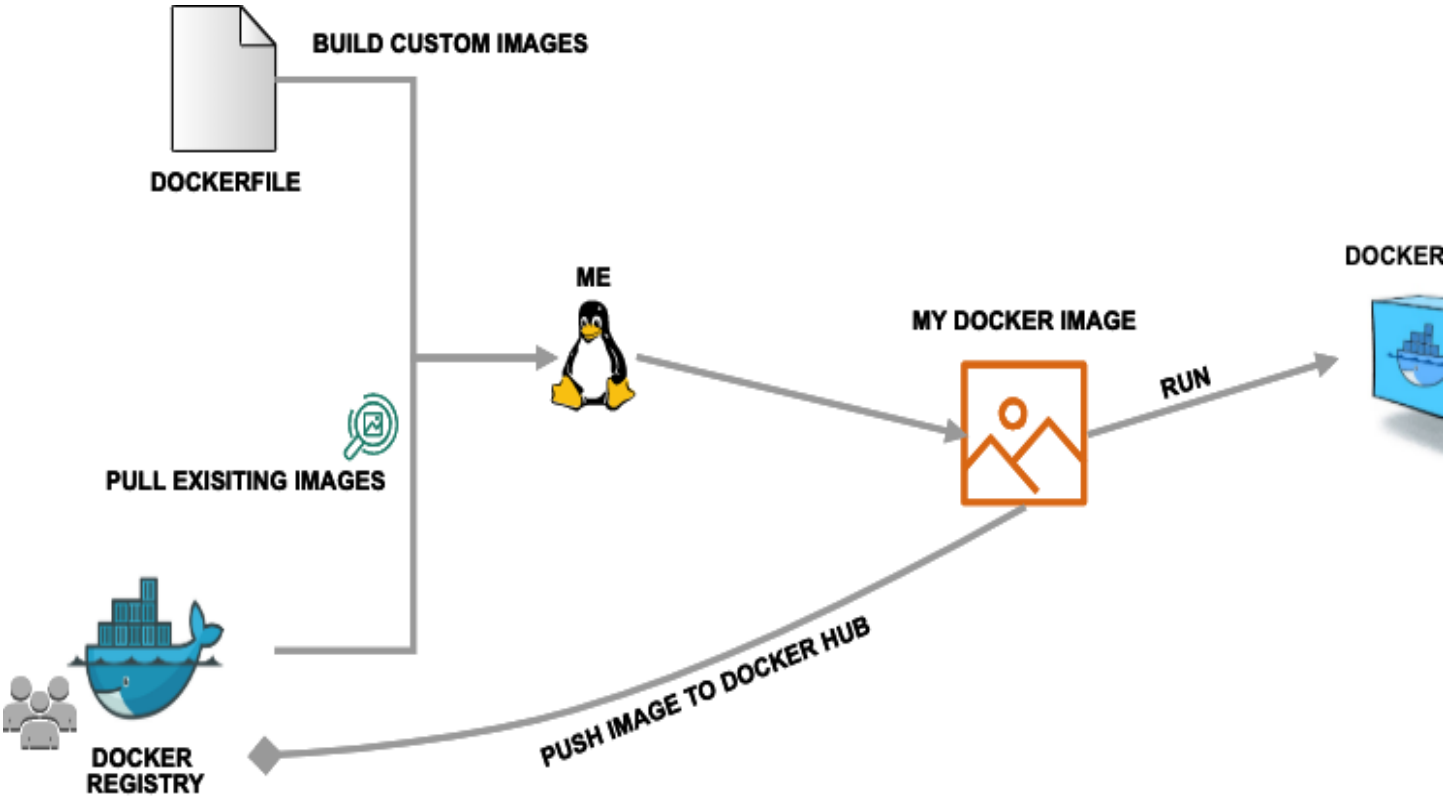
```
$ tacc/gateways19:0.1
```

would refer to the gateways19 image within the "tacc" repository and have a tag of "0.1".

TACC maintains multiple repositories on the Docker Hub including:

- tacc
- taccsciapps
- tapis
- abaco

Docker Worflow Overview



1.2 Getting Started With Docker

1.2.1 Prerequisites

- Install Docker on your laptop:
  - Mac

  - Windows
  - Ubuntu

To check if the installation was successful, open up your favorite Terminal (Mac,Linux) or the Docker Terminal (Windows) and try running

```
$ docker version
Client: Docker Engine - Community
Version:           19.03.5
API version:       1.40
Go version:        go1.12.12
Git commit:        633a0ea
Built:             Wed Nov 13 07:22:34 2019
OS/Arch:           darwin/amd64
Experimental:      false

Server: Docker Engine - Community
Engine:
Version:          19.03.5
API version:      1.40 (minimum version 1.12)
Go version:       go1.12.12
Git commit:       633a0ea
Built:            Wed Nov 13 07:29:19 2019
OS/Arch:          linux/amd64
Experimental:     false
containerd:
Version:          v1.2.10
GitCommit:␣
↪       b34a5c8af56e510852c35414db4c1f4fa6172339
runc:
Version:          1.0.0-rc8+dev
GitCommit:␣
↪       3e425f80a8c931f88e6d94a8c831b9d5aa481657
docker-init:
Version:          0.18.0
GitCommit:        fec3683
```

This also ensures you can access the docker daemon.

- Create a Docker Hub account

  Having a Docker Hub account makes it easier to share your containers with other researchers.

  Let's login into Docker Hub to be able to push images to your repository.

```
$ docker login
# Enter username/password
```

- Create a TACC Account

Note: If you do not have Docker installed on your laptop, you could also use https://training.play-with-docker.com/beginner-linux/

Working with Docker Hub images

- Say *hello* from Docker

  Let's run a simple hello-world container using the * docker run * command

```
$ docker run hello-world
Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world
1b930d010525: Pull complete
Digest:␣
↪sha256:5f179596a7335398b805f036f7e8561b6f0e32cd30a32f5e19d17a3cda6cc33d
Status: Downloaded newer image for hello-world:latest

Hello from Docker!
This message shows␣
↪that your installation appears to be working correctly.

To␣
↪generate this message, Docker took the following steps:
1. The Docker client contacted the Docker daemon.
2. The Docker daemon␣
↪pulled the "hello-world" image from the Docker Hub.
 (amd64)
 3. The Docker daemon␣
↪created a new container from that image which runs the
 executable␣
↪that produces the output you are currently reading.
 4. The Docker daemon streamed␣
↪that output to the Docker client, which sent it
 to your terminal.
```

  This simple command pulls the hello-world image from Docker Hub and prints the message.

- To pull an image off Docker Hub use the *docker pull* command

  Let's make this easier by first pulling the image from Docker Hub

```
$ docker pull hello-world:latest
latest: Pulling from library/hello-world
1b930d010525: Pull complete
Digest:␣
↪sha256:4df8ca8a7e309c256d60d7971ea14c27672fc0d10c5f303856d7bc48f8cc17ff
Status: Downloaded newer image for hello-world:latest
docker.io/library/hello-world:latest
```

- To verify the images are now available on your local machine, try:

```
$ docker images
REPOSITORY               TAG                 ␣
↪           IMAGE ID            CREATED             SIZE
hello-world              latest              ␣
↪         fce289e99eb9        11 months ago       1.84kB
```

- Now run the container:

```
$ docker run hello-world:latest
Hello from Docker!
This message shows␣
↪that your installation appears to be working correctly.
```

Note: As indicated in the output, if no tag is specified the "latest" tag is pulled.

You could also try:

```
$ docker inspect hello-world
# shows more information about container image
```

Docker Core Commands

Command	Usage
docker login	Authenticate to Docker Hub using username and password
docker run	*Usage: docker run -it user/imagetag*
docker build	*Usage: docker build -t username/imagetag .* Build a docker image from a Dockerfile in the current working directory. -t to tag the image
docker images	List all images on the local machine
docker tag	Add a new tag to an image
docker pull	Download an image from Docker Hub
docker push	*Usage: docker push username/imagetag* Upload an image to Docker Hub
docker inspect	*Usage: docker inspect containerID* Provide detailed information on constructs controlled by Docker
docker ps -a	List all containers on your system
docker rmi	Delete an image
docker rm	*Usage: docker rm -f [container name or ID]* Delete a container -f optional to remove running container
docker stop	*Usage: docker stop [container name or ID]* Stop a container

Getting more help with Docker

- The command line tools are very well documented:

```
$ docker --help
# shows all docker options and summaries
```

```
$ docker COMMAND --help
# shows options and summaries for a particular command
```

- Learn more about docker

1.3 Working with Docker

In this section, we will learn to create and use our own docker container.

Note: Prerequisites: 1. Have Docker installed on your laptop 2. Create a Docker Hub Account

1.3.1 Building Images From a *Dockerfile*

- Dockerfiles are a reproducible and well documented method of developing your own container.
- They store the whole procedure of how an image is built.

Dockerfile Format

A Dockerfile has two type of fields:

- Instructions followed by arguments and comments
- A basic Dockerfile looks like

```
# Comment
INSTRUCTION arguments
```

General Steps

1. Choose a base operating system
2. Install dependencies and other useful packages
3. Install scientific application
4. Set any environment variable that might be useful

1.3.2 Install a tool in a Docker container

Let's build a container that will run fastqc a popular bioinformatics tool for checking the quality of DNA sequencing reads.

To make it easier, follow along with the sample Dockerfile.

```
$ cat Dockerfile
```

```
# Choose a base operating system
FROM ubuntu:18.04

# Update and install necessary packages
RUN apt-get update && apt-get upgrade -y \
  && apt-get␣
↪install -y wget unzip default-jdk libfindbin-libs-perl

# Install the application
RUN wget https://www.bioinformatics.
↪babraham.ac.uk/projects/fastqc/fastqc_v0.11.8.zip \
    && unzip fastqc_v0.11.8.zip \
    && rm fastqc_v0.11.8.zip \
```

(continues on next page)

(continued from previous page)

```
    && chmod 755 /FastQC/fastqc

# Use environment variable to add executable to PATH
ENV PATH "/FastQC:$PATH"
```

From your current working directory, preferably a clean one, copy the contents of this file into a new file called Dockerfile and save it.

Build

```
$ docker build -t username/fastqc:0.11.8 .
Sending build context to Docker daemon  251.4MB
Step 1/4 : FROM ubuntu:18.04
---> 775349758637
. . .
Successfully built b5d705fbdfa1
Successfully tagged reshg/fastqc:0.11.8
```

Check images

```
$ docker images
REPOSITORY                      TAG
↪            IMAGE ID            CREATED              SIZE
reshg/fastqc                    0.11.8
↪           b5d705fbdfa1        3 hours ago         708MB
```

Run

We use the **docker run** command to run containers from an image. We pass a command to run in the container. Similar to running other programs on Unix systems, we can run containers in the foreground (attached) or in the background.

```
$ docker run username/fastqc:0.11.8 which fastqc
/FastQC/fastqc
```

Unpacking the 'docker run' command

docker run	Run something
–rm	Remove the container when the process completes
username/fastqc:0.11.8	The name of the container
which fastqc	The command to run

Push Image to Docker hub

```
$ docker push username/fastqc:0.11.8
```

1.3.3 Alternatively, you could also do this *interactively*

Note: Preferred way to build a docker image is by using Dockerfile. For the purpose of testing, working inside the container is sometimes helpful.

Open a base Docker Image

```
$ docker run --rm -it ubuntu /bin/bash
```

Unpacking the interactive 'docker run' command

docker run	Run something
–rm -it	Remove the container when the process completes and connect your terminal to the container runtime
ubuntu	The name of the container
/bin/bash	The type of shell to start

Install your tool in the image

```
root@ded8d40f1a1e:/#
# install dependencies
$ apt-get update && apt-get upgrade -y
$ apt-get␣
↪install -y wget unzip default-jdk libfindbin-libs-perl

# install FastQC
$ wget https://www.bioinformatics.
↪babraham.ac.uk/projects/fastqc/fastqc_v0.11.8.zip
$ unzip fastqc_v0.11.8.zip
$ rm fastqc_v0.11.8.zip

# make fastqc executable
$ chmod 755 /FastQC/fastqc

# add fastqc to the system path by linking to /bin
$ ln -s /FastQC/fastqc /bin
$ exit
```

Commit your image

```
$ docker ps -a
CONTAINER ID        IMAGE                                                  ␣
↪    COMMAND                  CREATED              STATUS␣
↪                      PORTS                     NAMES
9f0d7afff313␣
↪        ubuntu                             "/bin/bash"␣
↪              9 minutes ago       Exited (0) 18 seconds␣
↪ago                              affectionate_einstein

# Grab the CONTAINER␣
↪ID of the ubuntu image created just few minutes ago.
$ docker commit CONTAINER ID username/fastqc:0.11.8
sha256:738f35b39c5711f722cc6d9b550215454f2a7ea765c73667355d383a8a9285bf

$ docker images
REPOSITORY                      TAG                 ␣
↪           IMAGE ID            CREATED              SIZE
reshg/fastqc                    0.11.8              ␣
↪          738f35b39c57        12 seconds ago      718MB
```

Push your image to Docker Hub

```
$ docker push username/fastqc:0.11.8
The push refers to repository [docker.io/reshg/fastqc]
```

(continues on next page)

(continued from previous page)

```
6750c6c8d397: Pushing [==========>
↪                                    ]  124.3MB/654.2MB
```

Running a Container in Daemon mode

We can also run a container in the background. We do so using the -d flag:

```
$ docker run -d ubuntu sleep infinity
f406f6b0c34d4bba552a7106e951a5d667dcbfddcb429e2d42b0ac7a10a919fc

$ docker ps
CONTAINER ID
↪      IMAGE               COMMAND             CREATED
↪           STATUS              PORTS               NAMES
f406f6b0c34d        ubuntu
↪              "sleep infinity"    6 seconds ago
↪Up 5 seconds                               romantic_wilson

$ docker ps -a
CONTAINER ID        IMAGE
↪                    COMMAND             CREATED
↪     STATUS                      PORTS               NAMES
f406f6b0c34d        ubuntu
↪          "sleep infinity"    15 seconds ago      Up 14
↪seconds                                    romantic_wilson
b7c50065ea75        ubuntu
↪    "/bin/bash"         21 hours ago        Up 21 hours
↪                                  charming_robinson
a197e85bee14        reshg/fastqc:latest
↪     "/bin/bash"         21 hours ago        Exited (0)
↪21 hours ago                       reverent_williamson
4eb4cf433d32        reshg/fastqc:latest
↪         "which fastqc"      21 hours ago        Exited
↪(1) 21 hours ago                           stoic_dhawan
1eb1de6ac64c        reshg/fastqc
↪         "which fastqc"      21 hours ago        Exited
↪(1) 21 hours ago                           upbeat_mcnulty
```

Note: The docker ps command only shows you running containers - it does not show you containers that have exited. In order to see all containers on the system use docker ps -a.

Summary

A Dockerfile allows you to transparently document all the dependancies and steps needed to describe a software tool. You can then run this tool as a Docker container for full reproducibility.

CHAPTER 2

Containerize Your Code

Scenario: You are a researcher who has developed some new code for a scientific application. You now want to distribute that code for others to use in what you know to be a stable production environment (including OS and dependency versions). End users may want to use this code on their local workstations or on an HPC cluster.

In this section, we will build a container from scratch with some sample python code (provided). Please feel free to substitute in one of your own favorite programs or projects you are working on.

2.1 Install Code Interactively

In this section, we will explore the first half of a typical development workflow: installing an application interactively within a running docker container.

Note: Prerequisites: You should have access to a terminal with Docker installed.

2.1.1 Set Up

Before we begin, make a new directory somewhere on your local computer, and create an empty Dockerfile inside of it. It is important to carefully consider what files and folders are in the same *PATH* as a Dockerfile (known as the 'build context'). The *docker build* process will index and send all files and folders in the same directory as the Dockerfile to the Docker daemon, so take care not to *docker build* at a root level. For example:

```
$ cd ~/
$ mkdir python-container/
$ cd python-container/
$ touch Dockerfile
$ pwd
/Users/username/python-container/
$ ls
Dockerfile
```

Next, grab a copy of the source code we want to containerize:

```
#!/usr/bin/env python3
from random import random as r
from math import pow as p
from sys import argv

# Make sure number of attempts is given on command line
assert len(argv) == 2
attempts = int(argv[1])
inside=0
tries=0
ratio=0.

# Try the specified number of random points
while (tries < attempts):
    tries += 1
    if (p(r(),2) + p(r(),2) < 1):
        inside += 1

# Compute and print a final ratio
ratio=4.*(inside/(tries))
print("Final pi estimate from", attempts, "attempts =", ratio)
```

You can cut and paste the code block above into a new file called, e.g., *pi.py*, or download it from the following link:

https://raw.githubusercontent.com/TACC/containers_at_tacc/master/docs/scripts/pi.py

Now, you should have two files and nothing else in this folder:

```
$ pwd
/Users/username/python-container/
$ ls
Dockerfile     pi.py
```

2.1.2 Important Considerations

The most reproducible way to build a container is via a Dockerfile. We looked at pre-formed Dockerfiles in the first section of this workshop. But, now we want to build a Dockerfile from scratch. The questions you must ask yourself when starting a new Dockerfile include:

1. What is an appropriate base image?
2. What dependencies are required for my program?
3. What is the install process for my program?
4. What environment variables may be important?

2.1.3 Start an Interactive Session

Let's work through these questions by performing an **interactive installation** of our python script. In our hypothetical scenario, let's say our development platform / lab computer is a Linux workstation with Ubuntu 18.04. We know our code works on that workstation, so that is how we will containerize it. Use *docker run* to interactively attach to a fresh Ubuntu 18.04 container:

```
$ docker run -it -v $PWD:/code ubuntu:18.04 /bin/bash
```

Here is an explanation of the options:

```
docker run      # run a container
-it             # interactively attach terminal to inside of container
-v $PWD:/code   # mount the current directory to /code
unbuntu:18.04   # image and tag from Docker Hub
/bin/bash       # shell to start inside container
```

If this is your first time calling an Ubuntu 18.04 container on your laptop, then Docker will first download the image. The command prompt will change, signaling you are now 'inside' the container.

2.1.4 Update and Upgrade

The first thing we will typically do is use the Ubuntu package manager *apt* to update the list of available packages and install newer versions of the packages we have. We can do this with:

```
root@56c60cac8833:/# apt-get update
...
root@56c60cac8833:/# apt-get upgrade
...
```

Note: On the second command, you may need to choose 'Y' to install the upgrades.

2.1.5 Install Required Packages

For our python script to work, we need to install python3:

```
root@56c60cac8833:/# apt-get install python3
...
root@56c60cac8833:/# python3 --version
Python 3.6.9
```

An important question to ask is: Does this version match the version you are developing with on your local workstation? If not, make sure to install the correct version of python.

2.1.6 Install and Test Your Code

Since we are using a simple python script, there is not a difficult install process. However, we can make it executable, make sure it is in the user's PATH, and make sure it works as expected:

```
root@56c60cac8833:/# cd /code
root@56c60cac8833:/# chmod +rx pi.py
root@56c60cac8833:/# export PATH=/code:$PATH
```

Now test with the following:

```
root@56c60cac8833:/# cd /home
root@56c60cac8833:/# which pi.py
/code/pi.py
```

(continues on next page)

(continued from previous page)

```
root@56c60cac8833:/# pi.py 1000000
Final pi estimate from 1000000 attempts = 3.142804
```

2.1.7 Wrapping Up

We have a functional installation of *pi.py*! Now might be a good time to type *history* to see a record of the build process. When you are ready to start working on a Dockerfile, type *exit* to exit the container.

2.1.8 Hands On Exercise

What (if any) Docker images do you currently have on your machine? What (if any) Docker processes are currently running? If you have an Ubuntu base image, try removing it.

2.2 Build from a Dockerfile

After going through the build process interactively, we can translate our build steps into a Dockerfile using the directives described below.

Note: Prerequisites: You should have access to a terminal with Docker installed You should also have a copy of pi.py

---

2.2.1 The FROM Instruction

We can use the FROM instruction to start our new image from a known base image. This should be the first line of our Dockerfile. In our hypothetical scenario, we said our development platform / lab computer is a Linux workstation with Ubuntu 18.04. We know our code works on that workstation, so that is how we will containerize it. We will start our image from an official Ubuntu 18.04 image:

```
FROM ubuntu:18.04
```

Base images typically take the form *os:version*. Avoid using the '*latest*' version; it is hard to track where it came from and the identity of '*latest*' can change.

Tip: Browse Docker Hub to discover other potentially useful base images. Keep an eye out for the 'Official Image' badge.

2.2.2 The RUN Instruction

We can install updates, install new software, or download code to our image by running commands with the RUN instruction. In our case, our only dependency was Python3. So, we will use a RUN instruction and the Ubuntu package manager (*apt*) to install it. Keep in mind that the the *docker build* process cannot handle interactive prompts, so we use the *-y* flag with *apt*. We also need to be sure to update our apt packages. A typical RUN instruction for an Ubuntu base image may look like:

```
RUN apt-get update
RUN apt-get upgrade -y
RUN apt-get install -y python3
```

Each RUN instruction creates an intermediate image (called a 'layer'). Too many layers makes the Docker image less performant, and makes building less efficient. We can minimize the number of layers by combining the RUN instructions:

```
RUN apt-get update && apt-get upgrade -y && apt-get install -y python3
```

A similar RUN instruction for a RedHat / CentOS base image may look like:

```
RUN yum update -y && yum install -y python3
```

2.2.3 The COPY Instruction

There are a couple different ways to get your source code inside the image. One way is to use a RUN instruction with *wget* to pull your code from the web. When you are developing, however, it is usually more practical to copy code in from the Docker build context using the COPY instruction. For example, we can add our *pi.py* python script to a root-level */code* directory with the following instruction:

```
COPY pi.py /code/pi.py
```

And, don't forget to perform one more RUN instruction to make the script executable:

```
RUN chmod +rx /code/pi.py
```

2.2.4 The ENV Instruction

Another useful instruction is the ENV instruction. This allows the image developer to set environment variables inside the container runtime. In our interactive build, we added the */code* folder to the *PATH*. We can do this with an ENV instruction as follows:

```
ENV PATH "/code:$PATH"
```

2.2.5 Putting It All Together

The contents of the final Dockerfile should look like:

```
1 FROM ubuntu:18.04
2
3 RUN apt-get update && apt-get upgrade -y && apt-get install -y python3
4
5 COPY pi.py /code/pi.py
6
7 RUN chmod +rx /code/pi.py
8
9 ENV PATH "/code:$PATH"
```

2.2.6 Build the Image

Once the Dockerfile is written and we are satisfied that we have minimized the number of layers, the next step is to build an image. Building a Docker image generally takes the form:

```
$ docker build -t username/code:version .
```

The *-t* flag is used to name or 'tag' the image with a descriptive name and version. Optionally, you can preface the tag with your Docker Hub username. Adding that namespace allows you to push your image to a public container registry and share it with others. The trailing dot '.' in the line above simply indicates the location of the Dockerfile (a single '.' means 'the current directory').

To build our image, use:

```
$ docker build -t username/pi-estimator:0.1 .
```

Note: Don't forget to replace 'username' with your Docker Hub username.

2.2.7 Find the Image

Use *docker images* to ensure you see a copy of your image has been built. You can also use *docker inspect* to find out more information about the image.

```
$ docker images
REPOSITORY                TAG                 IMAGE ID            CREATED
↪SIZE
username/pi-estimator     0.1                 482bd4f0bc9b        14 minutes ago
↪200MB
ubuntu                    18.04               72300a873c2c        11 days ago
↪64.2MB
```

```
$ docker inspect username/pi-estimator:0.1
```

If you need to rename your image, you can either re-tag it with *docker tag*, or you can remove it with *docker rmi* and build it again. Issue each of the commands on an empty command line to find out usage information.

2.2.8 Test the Image

We can test a newly-built image two ways: interactively and non-interactively. In interactive testing, we will use *docker run* to start a shell inside the image, just like we did when we were building it interactively. The difference this time is that we are NOT mounting the code inside with the *-v* flag, because the code is already in the container:

```
[local]$ docker run --rm -it username/pi-estimator:0.1 /bin/bash
...
root@e01e374d7749:/# ls /code
pi.py
root@e01e374d7749:/# pi.py 1000000
Final pi estimate from 1000000 attempts = 3.137868
```

Here is an explanation of the options:

```
docker run      # run a container
--rm            # remove the container when we exit
-it             # interactively attach terminal to inside of container
username/...    # image and tag on local machine
/bin/bash       # shell to start inside container
```

Next, exit the container and test the code non-interactively. Notice we are calling the container again with *docker run*, but instead of specifying an interactive (*-it*) run, we just issue the command as we want to call it ('*pi.py 1000000*') on the command line:

```
$ docker run --rm username/pi-estimator:0.1 pi.py 1000000
Final pi estimate from 1000000 attempts = 3.141208
```

If there are no errors, the container is built and ready to share!

2.2.9 Hands On Exercise

Use *docker inspect* to look at the metadata for your *pi-estimator* image. Is the */code* folder in the *$PATH*? Determine the contents of *$PATH* inside the container to confirm.

2.3 Share Your Docker Image

Note: Prerequisites: You should have access to a terminal with Docker installed, a Docker Hub account, and a GitHub account. You should also have a copy of pi.py

Now that you have containerized, tested, and tagged your code in a Docker image, the next step is to disseminate it so others can use it.

2.3.1 Commit to GitHub

In the spirit of promoting Reproducible Science, it is now a good idea to create a new GitHub repository for this project and commit our files. The steps are:

1. Log in to GitHub and create a new repository called *pi-estimator*
2. Do not add a README or license file at this time
3. Then in your working folder, issue the following:

```
$ pwd
/Users/username/python-container/
$ ls
Dockerfile     pi.py
$ git init
$ git add *
$ git commit -m "first commit"
$ git remote add origin https://github.com/username/pi-estimator.git
$ git push -u origin master
```

Make sure to use the GitHub URL which matches your username and repo name. Let's also tag the repo as '0.1' to match our Docker image tag:

```
$ git tag -a 0.1 -m "first release"
$ git push origin 0.1
```

Finally, navigate back to your GitHub repo in a web browser and make sure your files were uploaded and the tag exists.

2.3.2 Push to Docker Hub

Docker Hub is the de facto place to share an image you built. Remember, the image must be name-spaced with either your Docker Hub username or a Docker Hub organization where you have write privileges in order to push it:

```
$ docker login
...
$ docker push username/pi-estimator:0.1
```

You and others will now be able to pull a copy of your container with:

```
$ docker pull username/pi-estimator:0.1
```

GitHub also has integrations to automatically update your image in the public container registry every time you commit new code.

For example, see: Set up automated builds

Note: After the next hands-on exercise, we will set up the GitHub-Docker integration

2.3.3 Hands-On Exercise

Scenario: You have the great idea to update your python code to use *argparse* to better handle the command line arguments. Outside of the container, modify *pi.py* to look like:

```
#!/usr/bin/env python3
from random import random as r
from math import pow as p
from sys import argv

# Use argparse to take command line options and generate help text
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("number", help="number of random points (int)", type=int)
args = parser.parse_args()

# Grab number of attempts from command line
attempts = args.number
inside=0
tries=0
ratio=0.

# Try the specified number of random points
while (tries < attempts):
    tries += 1
    if (p(r(),2) + p(r(),2) < 1):
        inside += 1

```

(continues on next page)

(continued from previous page)

```
# Compute and print a final ratio
ratio=4.*(inside/(tries))
print("Final pi estimate from", attempts, "attempts =", ratio)
```

(New and modified lines are highlighted). With this change, the user can execute '*pi.py -h*' to get usage information. You can also download this code from here:

https://raw.githubusercontent.com/TACC/containers_at_tacc/master/docs/scripts/pi-updated.py

Next, update the Dockerfile to include a new kind of instruction at the very end of the file - CMD:

```
FROM ubuntu:18.04

RUN apt-get update && apt-get upgrade -y && apt-get install -y python3

COPY pi.py /code/pi.py

RUN chmod +x /code/pi.py

ENV PATH "/code:$PATH"

CMD ["pi.py", "-h"]
```

This command will be executed in the container if the user calls the container without any arguments.

Finally, rebuild the container and update the version tag to '0.2'. Test that the code in the new container has been updated, and that it is working as expected.

Solution

```
# 1: edit pi.py to include new code
# 2: edit Dockerfile to include CMD instruction
# 3: follow the steps below

$ docker build -t username/pi-estimator:0.2 .
$ docker run --rm username/pi-estimator:0.2
usage: pi.py [-h] number

positional arguments:
  number      number of random points (int)

optional arguments:
  -h, --help  show this help message and exit

$ docker run --rm username/pi-estimator:0.2 pi.py 1000000
Final pi estimate from 1000000 attempts = 3.143672

$ docker run --rm -it username/pi-estimator:0.2 /bin/bash
root@c5aa145e5546:/# which pi.py
/code/pi.py
root@c5aa145e5546:/# pi.py 1000000
Final pi estimate from 1000000 attempts = 3.141168
root@c5aa145e5546:/# exit

$ docker push username/pi-estimator:0.2
```

2.3.4 Set up a GitHub-Docker Hub Integration

Rather than commit to GitHub AND push to Docker Hub each time you want to release a new version, you can set up an integration between the two services that automates it. The key benefit is you only have to commit to one place (GitHub), and you know the image available on Docker Hub is always in sync.

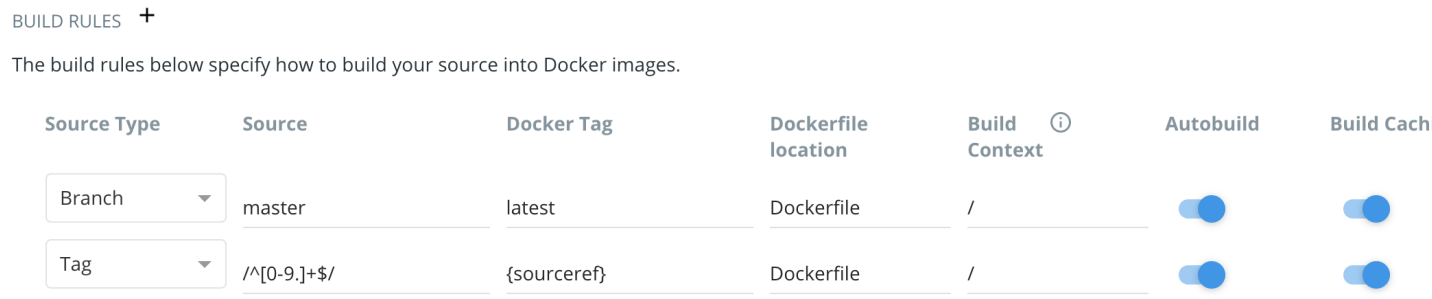
To set up the integration, navigate to your new Docker repository in a web browser, which should be at an address similar to:

https://hub.docker.com/repository/docker/YOUR-DOCKER-USERNAME/pi-estimator

Click on Builds => Link to GitHub. (If this is your first time connecting a Docker repo to a GitHub repo, you will need to set it up. Press the 'Connect' link to the right of 'GitHub'. If you are already signed in to both Docker and GitHub in the same browser, it takes about 15 seconds to set up).

Once you reach the Build Configurations screen, you will select your GitHub username and repository named pi-estimator.

Leaving all the defaults selected will cause this Docker image to rebuild every time you push code to the master branch of your GitHub repo. For this example, set the build to to trigger whenever a new release is tagged:



Click 'Save and Build' and check the 'Timeline' tab on Docker Hub to see if it is working as expected.

2.3.5 Commit to GitHub (Again)

Finally, push your modified code to GitHub and tag the release as 0.2 to trigger another automated build:

```
$ git add *
$ git commit -m "using argparse to parse args"
$ git push
$ git tag -a 0.2 -m "release version 0.2"
$ git push origin 0.2
```

By default, the git push command does not transfer tags, so we are explicitly telling git to push the tag we created (0.2) to the remote (origin).

Now, check the online GitHub repo to make sure your change / tag is there, and check the Docker Hub repo to see if your image is automatically rebuilding.

2.3.6 Other Considerations

The best way to learn to build docker images is to practice building lots of images for tools you use. The online Docker documentation contains a lot of good advice on building images.

Some miscellaneous tips for building images include:

- Save your Dockerfiles – GitHub is a good place for this

- You probably don't want to use ENTRYPOINT - turns an container into a black box
- If you use CMD, make it print the help text for the containerized code
- Usually better to use COPY instead of ADD
- Order of operations in the Dockerfile is important; combine steps where possible
- Remove temporary and unnecessary files to keep images small
- Avoid using *latest* tag; use explicit tag callouts
- The command *docker system prune* will help free up space in your local environment
- Use *docker-compose* for multi-container pipelines and microservices
- A good rule of thumb is one tool or process per container

CHAPTER 3

Containers on High Performance Compute Clusters

High performance computing (HPC) systems serve a large role in academic computing at scale. In this portion of the training, we will explore methods for running containers that you develop on HPC systems and also discovering containers built by the community that you can utilize.

Objectives for this session

- Use Singularity to execute Docker containers on a HPC system
- Create a container that can use the Message Passing Interface (MPI) to use multiple nodes in parallel
- Understand how to run containers that use GPUs for computation
- Discover community curated software containers available at TACC

3.1 Introduction to Singularity

Note:

Prerequisites This section uses the Stampede2 compute cluster to run Singularity. An active allocation on Stampede2 is required, though most content will apply to any system that supports Singularity.

At face value, Singularity is an alternative container implementation to Docker that has an overlapping set of features but some key differences as well. Singularity is commonly available on shared clusters, such as TACC's HPC systems, because the Docker runtime is not secure on systems where users are not allowed to have "escalated privileges". Importantly, the Singularity runtime is compatible with Docker containers! So in general, we follow the practice of using Docker to develop containers and using Singularity simply as a runtime to execute containers on HPC systems.

If you are familiar with Docker, Singularity will feel familiar.

3.1.1 Login to Stampede2

For today's training, we will use the Stampede2 supercomputer, the 18th most powerful system in the world at the time of the course. To login, you need to establish a SSH connection from your laptop to the Stampede2 system. Instructions depend on your laptop's operating system.

Mac / Linux:

Open the application 'Terminal'
ssh username@stampede2.tacc.utexas.edu
(enter password)
(enter 6-digit token)

Windows:

If using Windows Subsystem for Linux, use the Mac / Linux instructions.
If using an application like 'PuTTY'
enter Host Name: stampede2.tacc.utexas.edu
(click 'Open')
(enter username)
(enter password)
(enter 6-digit token)

When you have successfully logged in, you should be greeted with some welcome text and a command prompt.

3.1.2 Start an Interactive Session

The Singularity module is currently only available on compute nodes at TACC. To use Singularity interactively, start an interactive session on a compute node using the `idev` command.

```
$ idev -m 40
```

If prompted to use a reservation, choose yes. Once the command runs successfully, you will no longer be on a login node, but instead have a shell on a dedicated compute node.

3.1.3 Load the Singularity Module

By default, the `singularity` command is not visible, but it can be added to the environment by loading the module.

```
$ module list

$ module spider singularity

$ module load tacc-singularity

$ module list
```

Now the singularity command is available.

```
$ type singularity

$ singularity help
```

3.1.4 Core Singularity Commands

Pull a Docker container

Containers in the Docker registry may be downloaded and used, assuming the underlying architecture (e.g. x86) is the same between the container and the host.

```
$ singularity pull docker://godlovedc/lolcow

$ ls
```

There may be some warning messages, but this command should download the latest version of the "lolcow" container and save it in your current working directory as `lolcow_latest.sif`.

Interactive shell

The `shell` command allows you to spawn a new shell within your container and interact with it as though it were a small virtual machine.

```
$ singularity shell lolcow_latest.sif

Singularity lolcow_latest.sif:~>
```

The change in prompt indicates that you have entered the container (though you should not rely on that to determine whether you are in container or not).

Once inside of a Singularity container, you are the same user as you are on the host system. Also, a number of host directories are mounted by default.

```
Singularity lolcow_latest.sif:~> whoami

Singularity lolcow_latest.sif:~> id

Singularity lolcow_latest.sif:~> pwd

Singularity lolcow_latest.sif:~> exit
```

Note: Docker and Singularity have very different conventions around how host directories are mounted within the container. In many ways, Singularity has a simpler process for working with data on the host, but it is also more prone to inadvertantly having host configurations "leak" into the container.

Run a container's default command

Just like with Docker, Singularity can run the default "entrypoint" or default command of a container with the `run` subcommand. These defaults are defined in the Dockerfile (or Singularityfile) that define the actions a container should perform when someone runs it.

```
    singularity run lolcow_latest.sif

 _________________________________________
< The time is right to make new friends. >
 -----------------------------------------
        \   ^__^
         \  (oo)\_______
            (__)\       )\/\
                ||----w |
                ||     ||
```

Note: You may receive a warning about "Setting locale failed". This is because, by default, Singularity sets all shell environment variables inside the container to match whatever is on the host. To override this behavior, add the `--cleanenv` argument to your command.

Executing arbitrary commands

The exec command allows you to execute a custom command within a container. For instance, to execute the `cowsay` program within the lolcow_latest.sif container:

```
    $ singularity exec --cleanenv lolcow_latest.sif cowsay Singularity runs Docker␣
↪containers on HPC systems
 ______________________________________
/ Singularity runs Docker containers on \
\ HPC systems                           /
 --------------------------------------
        \   ^__^
         \  (oo)\_______
            (__)\       )\/\
                ||----w |
                ||     ||
```

Note: `exec` also works with the library://, docker://, and shub:// URIs. This creates an ephemeral container that executes a command and disappears.

Once you are finished with your interactive session, you can end it and return to the login node with the exit command:

```
$ exit
```

3.2 Using HPC Environments

Note:

Prerequisites This section uses the Stampede2 compute cluster to run Singularity. An active allocation on Stampede2 is required, though most content will apply to any system that supports Singularity.

Conducting analyses on high performance computing clusters happens through very different patterns of interaction than running analyses on a VM or on your own laptop. When you login, you are on a node that is shared with lots of

people. Trying to run jobs on that node is not "high performance" at all. Those login nodes are just intended to be used for moving files, editing files, and launching jobs.

Most jobs on a HPC cluster are neither interactive, nor realtime. When you submit a job to the scheduler, you must tell it what resources you need (e.g. how many nodes, what type of nodes) and what you want to run. Then the scheduler finds resources matching your requirements, and runs the job for you when it can.

For example, if you want to run the command:

```
singularity exec docker://python:latest /usr/local/bin/python
```

On a HPC system, your job submission script would look something like:

```
#!/bin/bash

#SBATCH -J myjob                        # Job name
#SBATCH -o output.%j                    # Name of stdout output file (%j expands to␣
↪jobId)
#SBATCH -p normal                       # Queue name
#SBATCH -N 1                            # Total number of nodes requested (68 cores/
↪node)
#SBATCH -n 1                            # Total number of mpi tasks requested
#SBATCH -t 02:00:00                     # Run time (hh:mm:ss) - 4 hours

module load tacc-singularity
singularity exec docker://python:latest /usr/local/bin/python
```

This example is for the Slurm scheduler, a popular one used by all TACC systems. Each of the #SBATCH lines looks like a comment to the bash kernel, but the scheduler reads all those lines to know what resources to reserve for you.

Note: Every HPC cluster is a little different, but they almost universally have a "User's Guide" that serves both as a quick reference for helpful commands and contains guidelines for how to be a "good citizen" while using the system. For TACC's Stampede2 system, the user guide is at: https://portal.tacc.utexas.edu/user-guides/stampede2

3.2.1 How do HPC systems fit into the development workflow?

A couple of things to consider when using HPC systems:

1. Using 'sudo' is not allowed on HPC systems, and building a Singularity container from scratch requires sudo. That means you have to build your containers on a different development system, which is why we started this course developing Docker on your own laptop). You can pull a docker image on HPC systems.
2. If you need to edit text files, command line text editors don't support using a mouse, so working efficiently has a learning curve. There are text editors that support editing files over SSH. This lets you use a local text editor and just save the changes to the HPC system.

In general, most TACC staff that work with containers develop their code locally and then deploy their containers to HPC systems to do analyses at scale. If the containers are written in a way that accommodates the small differences between the Docker and Singularity runtimes, the transition is fairly seamless.

3.2.2 Differences between Docker and Singularity

Host Directories

Docker: None by default. Use `-v <source>:<destination>` to mount a source host directory to an arbitrary destination within the container.

Singularity: Mounts your current working directory, $HOME directory, and some system directories by default. Other defaults may be set in a system-wide configuration. The `--bind` flag is supported but rarely used in practice.

User ID

Docker: Defined in the Dockerfile, but containers run as root unless a different user is defined or specified on the command line. This user ID only exists within the container, and care must be taken when working with files on the host filesystem to make sure permissions are set correctly.

Singularity: Containers are run in "userspace", so you are the same user and user ID both inside and outside the container.

Image Format

Docker: Containers are stored in layers and managed in a repository by Docker. The `docker images` command will show you what containers are on your local machine and images are always referenced by their repository and tag name.

Singularity: Containers are files. Singularity can build a container on the fly if you specify a repository, but ultimately they are stored as individual files, with all the benefits and dangers inherent to files.

3.2.3 Running a Batch Job on Stampede2

If you are not already, please login to the Stampede2 system, just like we did at the start of the previous section. You should be on one of the login nodes of the system.

We will not be editing much text directly on Stampede2, but we need to do a little. If you have a text editor you prefer, use it for this next part. If not, the `nano` text editor is probably the most accessible for those new to Linux.

Create a file called "pi.slurm" on the work filesystem:

```
cd $WORK
mkdir containers-at-tacc
cd containers-at-tacc
nano pi.slurm
```

Those commands should open a new file in the nano editor. Either type in (or copy and paste) the following Slurm script.

```
#!/bin/bash

#SBATCH -J calculate-pi                 # Job name
#SBATCH -o output.%j                    # Name of stdout output file (%j expands to␣
↪jobId)
#SBATCH -p normal                       # Queue name
#SBATCH -N 1                            # Total number of nodes requested (68 cores/
↪node)
#SBATCH -n 1                            # Total number of mpi tasks requested
#SBATCH -t 00:10:00                     # Run time (hh:mm:ss)
#SBATCH --reservation Containers        # a reservation only active during the training
```

(continues on next page)

(continued from previous page)

```
module load tacc-singularity

echo "running the lolcow container:"
singularity run docker://godlovedc/lolcow:latest

echo "estimating the value of Pi:"
singularity exec docker://USERNAME/pi-estimator:0.1 pi.py 10000000
```

- Don't forget to replace `USERNAME` with your DockerHub username! If you didn't publish a pi-estimator container from the previous sections, you are welcome to use "wallen" as the username to pull Joe Allen's container.
- If you have more than one allocation, you will need to add another line specifying what allocation to use, such as: `#SBATCH -A AllocationName`

Once you are done, try submitting this file as a job to Slurm.

```
sbatch pi.slurm
```

You can check the status of your job with the command `showq -u`.

Once your job has finished, take a look at the output:

```
cat output*
```

If your containers ran successfully, then congratulations! While this was just a toy example, you have now gone through all the motions of a development lifecycle:

- capturing your code and requirements as a Docker recipe
- deploying your own code to run on your laptop and a HPC system
- using someone else's container both on your laptop and a HPC system
- publishing your code to DockerHub so that it can be shared with others

3.3 MPI and GPU Containers

3.3.1 Message Passing Interface (MPI) for running on multiple nodes

Distributed MPI execution of containers is supported by Singularity.

Since (by default) the network is the same inside and outside the container, the communication between containers usually just works. The more complicated bit is making sure that the container has the right set of MPI libraries to interact with high-speed fabrics. MPI is an open specification, but there are several implementations (OpenMPI, MVAPICH2, and Intel MPI to name three) with some non-overlapping feature sets. There are also different hardware implementations (e.g. Infiniband, Intel Omnipath, Cray Aries) that need to match what is inside the container. If the host and container are running different MPI implementations, or even different versions of the same implementation, MPI may not work.

The general rule is that you want the version of MPI inside the container to be the same version or newer than the host. You may be thinking that this is not good for the portability of your container, and you are right. Containerizing MPI applications is not terribly difficult with Singularity, but it comes at the cost of additional requirements for the host system.

Warning: Many HPC Systems, like Stampede2, have high-speed, low latency networks that have special drivers. Infiniband, Aries, and OmniPath are three different specs for these types of networks. When running MPI jobs, if the container doesn't have the right libraries, it won't be able to use those special interconnects to communicate between nodes. This means that MPI containers don't provide as much portability between systems.

Base Docker images

When running at TACC, we have a set of curated Docker images for use in the FROM line of your own containers. You can see a list of available images at https://github.com/TACC/tacc-containers

Image	Frontera	Stampede2	Maverick2	Local Dev
tacc/tacc-centos7-mvapich2.3-ib	X		X	X
tacc/tacc-centos7-mvapich2.3-psm2		X		
tacc/tacc-centos7-impi19.0.7-common	X*	X*	X*	X
tacc/tacc-ubuntu18-mvapich2.3-ib	X		X	X
tacc/tacc-ubuntu18-mvapich2.3-psm2		X		
tacc/tacc-ubuntu18-impi19.0.7-common	X*	X*	X*	X

Note: *Must be invoked with `singularity run` on HPC.

LS5 is not present because it uses a proprietary fabric library which cannot be built into a container.

In this tutorial, we will be using use the `tacc/tacc-ubuntu18-impi19.0.7-common` image to satisfy the MPI architectures on both Stampede2 and Frontera, while also allowing intra-node (single-node, multiple-core) testing on your local development system.

Building a MPI aware container

On your local laptop, go back to the directory where you built the "pi" Docker image and download (or copy and paste) two additional files:

- Dockerfile.mpi
- pi-mpi.py

Take a look at both files. `pi-mpi.py` is an updated Python script that uses MPI for parallelization. `Dockerfile.mpi` is an updated Dockerfile that uses the TACC base image to satisfy all the MPI requirements on Stampede2.

Note: A full MPI stack with `mpicc` is available in these containers, so you can compile code too.

With these files downloaded, we can build a new MPI-capable container for faster execution.

```
$ docker build -t USERNAME/pi-estimator:0.1-mpi -f Dockerfile.mpi .
```

Note: Don't forget to change USERNAME to your DockerHub username!

To prevent this build from overwriting our previous container (`USERNAME/pi-estimator:0.1`), the "tag" was changed from `0.1` to `0.1-mpi`. We also could have just renamed the "repository" to something like `USERNAME/pi-estimator-mpi:0.1`. You will see both conventions used on DockerHub. For different versions, or maybe

architectures, of the same codebase, it is okay to differentiate them by tag. Independent codes should use different repository names.

Once you have successfully built the image, push it up to DockerHub with the `docker push` command so that we can pull it back down on Stampede2.

```
$ docker push USERNAME/pi-estimator:0.1-mpi
```

Running an MPI Container Locally

Before using allocation hours at TACC, it's always a good idea to test your code locally. Since your local workstation may not have as many resources as a TACC compute node, testing is often done with a *toy* sized problem to check for correctness.

Our `pi-estimator:0.1-mpi` container started FROM `tacc/tacc-ubuntu18-impi19.0.7-common`, which is capable of locally testing the MPI capabilities using shared memory. Launch the `pi-mpi.py` script with `mpirun` from inside the container. By default, `mpirun` will launch as many processes as cores, but this can be controlled with the `-n` argument.

Lets try computing Pi with 10,000,000 samples using 1 and 2 processors.

```
# Run using 1 processor
$ docker run --rm -it USERNAME/pi-estimator:0.1-mpi \
        mpirun -n 1 pi-mpi.py 10000000

# Run using 2 processors
$ docker run --rm -it USERNAME/pi-estimator:0.1-mpi \
        mpirun -n 2 pi-mpi.py 10000000
```

You should notice that while the estimate stayed roughly the same, the execution time halved as the program scaled from one to two processors.

Note: If the computation time did not decrease, your Docker Desktop may not be configured to use multiple cores.

Now that we validated the container locally, we can take it to a TACC node and scale it up further.

Running an MPI Container on Stampede2

To start, lets allocate a single KNL node, which has 68 physical cores per node, but lets only use 8 cores to make the log messages a little more legible.

Running interactively

Please use `idev` to allocate this 8-task compute node.

```
$ idev -m 60 -p normal -N 1 -n 8
```

Once you have your node, pull the container and run it as follows:

```
# Load singularity module
$ module load tacc-singularity

# Change to $SCRATCH directory so containers don't go over your $HOME quota
```

(continues on next page)

(continued from previous page)

```
$ cd $SCRATCH

# Pull container
$ singularity pull docker://USERNAME/pi-estimator:0.1-mpi

# Run container sequentially
$ singularity run pi-estimator_0.1-mpi.sif pi-mpi.py 10000000
$ ibrun -n 1 singularity run pi-estimator_0.1-mpi.sif pi-mpi.py 10000000

# Run container distributed
$ ibrun singularity run pi-estimator_0.1-mpi.sif pi-mpi.py 10000000

# Run container with fewer tasks
$ ibrun -n 4 singularity run pi-estimator_0.1-mpi.sif pi-mpi.py 10000000
```

In our local tests, the **container** `mpirun` program was used to launch multiple processes, but this does not scale to multiple nodes. When using multiple nodes at TACC, you should always use `ibrun` to call singularity to launch a container per process across each **host**.

Note: The `*impi*` containers **must** be launched with `singularity run` on HPC systems.

TACC uses a command called `ibrun` on all of its systems that configures MPI to use the high-speed, low-latency network, and binds processes to specific cores. If you are familiar with MPI, this is the functional equivalent to `mpirun`.

Take some time and try running the program with more samples. Just remember that each extra digit will increase the runtime by about 10-times the previous, so hit `Ctrl-C` to terminate something that's taking too long.

Running via batch submission

To run a container via non-interactive batch job, the container should first be downloaded to a performant filesystem like `$SCRATCH` or `$HOME`.

```
$ idev -m 60 -p normal -N 1
$ cd $SCRATCH
$ module load tacc-singularity
$ singularity pull docker://USERNAME/pi-estimator:0.1-mpi
$ ls *sif
$ exit
```

After pulling the container, the image file can be referred to in an sbatch script. Please create `pi-mpi.sbatch` with the following text:

```
#!/bin/bash

#SBATCH -J calculate-pi-mpi           # Job name
#SBATCH -o calculate-pi-mpi.%j        # Name of stdout output file (%j expands to␣
↪jobId)
#SBATCH -p normal                     # Queue name
#SBATCH -N 1                          # Total number of nodes requested (68 cores/
↪node)
#SBATCH -n 8                          # Total number of mpi tasks requested
#SBATCH -t 00:10:00                   # Run time (hh:mm:ss)
```

(continues on next page)

(continued from previous page)

```
#SBATCH --reservation Containers      # a reservation only active during the training

module load tacc-singularity
cd $SCRATCH
ibrun singularity exec pi-estimator_0.1-mpi.sif pi-mpi.py 10000000
```

Then, you can submit the job with `sbatch`

```
$ sbatch pi-mpi.sbatch
```

Check the status of your job with `squeue`

```
$ squeue -u USERNAME
```

When your job is done, the output will be in `calculate-pi-mpi.[job number]`, and can be viewed with `cat`, `less`, or your favorite text editor.

Once done, try scaling up the program to two nodes (`-N 2`) and 16 tasks (`-n 16`) by changing your batch script or idev session. After that, try increasing the number of samples to see how accurate your estimate can get.

Note: If your batch job is running too long, you can finding the job number with *squeue -u [username]* and then terminate it with `scancel [job number]`

3.3.2 Singularity and GPU Computing

Singularity **fully** supports GPU utilization by exposing devices at runtime with the `--nv` flag. This is similar to `nvidia-docker`, so all docker containers with libraries that are compatible with the drivers on our systems can work as expected.

For instance, the latest version of caffe can be used on TACC systems as follows:

```
# Work from a compute node
$ idev -m 60 -p rtx

# Load the singularity module
$ module load tacc-singularity

# Pull your image
$ singularity pull docker://nvidia/caffe:latest

# Test the GPU
$ singularity exec --nv caffe_latest.sif caffe device_query -gpu 0
```

Note: If this resulted in an error and the GPU was not detected, and you are on a GPU-enabled compute node, you may have excluded the `--nv` flag.

As previously mentioned, the main requirement for GPU-enabled containers to work is that the version of the host drivers matches the major version of the library inside the container. So, for example, if CUDA 10 is on the host, the container needs to use CUDA 10 internally.

For a more exciting test, the latest version of Tensorflow can be benchmarked as follows:

```
# Change to your $SCRATCH directory
$ cd $SCRATCH

# Download the benchmarking code
$ git clone --branch cnn_tf_v2.1_compatible https://github.com/tensorflow/benchmarks.
↪git

# Pull the image
$ singularity pull docker://tensorflow/tensorflow:latest-gpu

# Run the code
$ singularity exec --nv tensorflow_latest-gpu.sif python \
    benchmarks/scripts/tf_cnn_benchmarks/tf_cnn_benchmarks.py \
    --num_gpus=4 --model resnet50 --batch_size 32 --num_batches 100
```

Try different numbers of gpus, batch sizes, and total batches to see how the parameters affect the benchmark.

Note: If the benchmark crashes you the batch may be too large for GPU memory, or you requested more GPUs than exist on the system.

CHAPTER 4

Indices and tables

- genindex
- modindex
- search