
Containers@TACC Documentation

Release 0.1

Texas Advanced Computing Center

Apr 11, 2023

Contents:

1	Welcome to Containers @ TACC!	3
1.1	Introduction to Containers	3
1.2	Getting Started With Docker	6
1.3	Working with Docker	10
2	Containerize Your Code	15
2.1	Install Code Interactively	15
2.2	Build from a Dockerfile	18
2.3	Share Your Docker Image	21
3	Containers on High Performance Compute Clusters	25
3.1	Introduction to Singularity/Apptainer	25
3.2	Using HPC Environments	28
3.3	MPI and GPU Containers	31
4	Indices and tables	39

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

Welcome to Containers @ TACC!

In this section, we will be introduced to containers, their uses, and different existing container technologies. Our focus would be on one such container technology called “Docker”.

Objectives for this session:

- Describe the essentials of containers
- Gain experience with the Docker command line interface
- Find and pull existing containers from Docker Hub
- Run containers interactively and non-interactively

1.1 Introduction to Containers

Containers are an important common currency for app development, web services, scientific computing, and more. Containers allow you to package an application along with all of its dependencies, isolate it from other applications and services, and deploy it consistently and reproducibly and *platform-agnostically*. In this introductory module, we will learn about containers and their uses, in particular the containerization platform **Docker**.

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 *reproducibility* by isolating the application from environment.

1.1.2 How is a Container Different from a VM?

Virtual machines enable application and resource isolation, run on top of a hypervisor (high overhead). Multiple VMs can run on the same physical infrastructure - from a few to dozens depending on resources. VMs take up more disk space and have long start up times (~minutes).

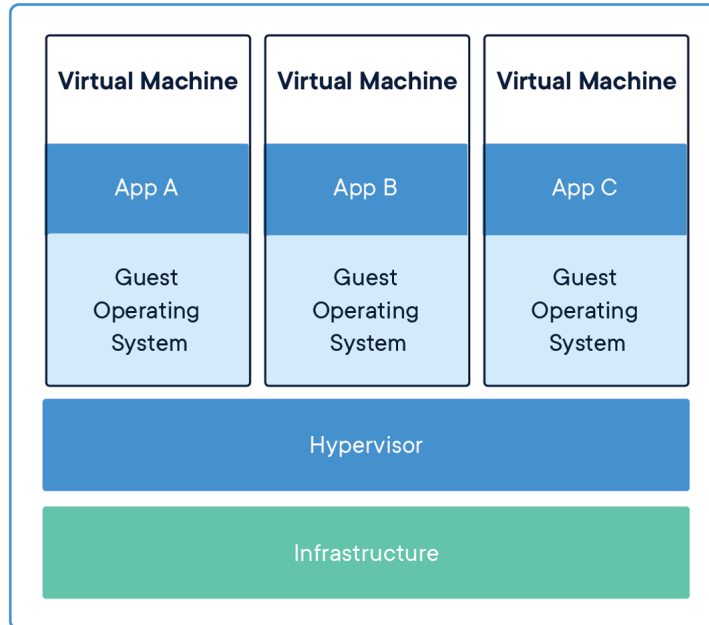


Fig. 1: Applications isolated by VMs.

Containers enable application and resource isolation, run on top of the host operating system. Many containers can run on the same physical infrastructure - up to 1,000s depending on resources. Containers take up less disk space than VMs and have very short start up times (~100s of ms).

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

1.1.3 Container Technologies

Docker

Docker is a containerization platform that uses OS-level virtualization to package software and dependencies in deliverable units called containers. It is by far the most common containerization platform today, and most other container platforms are compatible with Docker. (E.g. Singularity and Shifter are two containerization platforms you'll find in HPC environments).



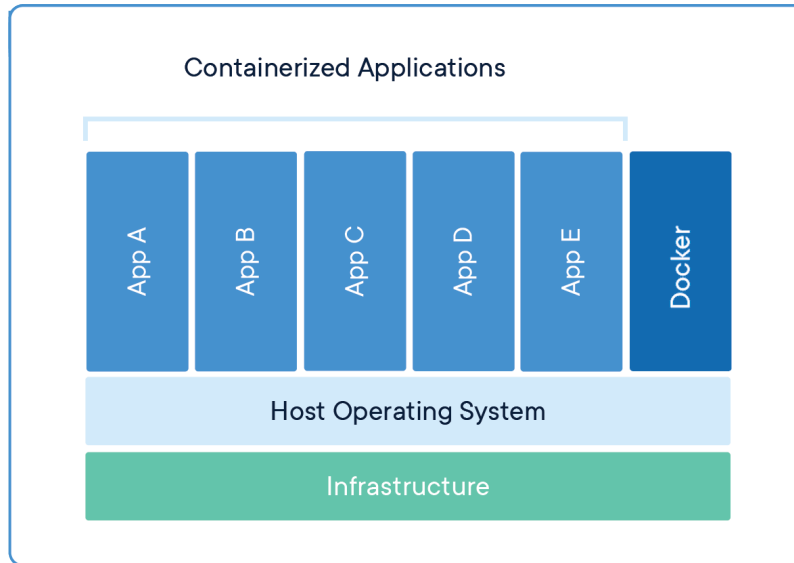


Fig. 2: Applications isolated by containers.

Singularity (now Apptainer)

Singularity (recently renamed as Apptainer) is a container solution designed to execute applications at bare-metal performance while being secure, portable, and 100% reproducible. Singularity's permissions model makes it a popular choice for shared HPC environments where Docker cannot be supported. It has its own syntax for building containers but also support pulling and running Docker containers.

In general we use **Docker** to develop new containers and run them on our laptops. We use **Singularity (Apptainer)** as a runtime on our HPC systems. We can find existing containers that are compatible with both Docker and Singularity (Apptainer) platforms (among others) at:

1. [Docker Hub](#)
2. [Quay.io](#)
3. [BioContainers](#)

1.1.4 Some Quick Definitions

Dockerfile

A Dockerfile is a recipe for creating a Docker image. It is a human-readable, plain text file that contains a sequential set of commands (*a recipe*) for installing and configuring an application and all of its dependencies. The Docker command line interface is used to interpret a Dockerfile and "build" an image based on those instructions. Other container build environments, such as Apptainer, have different syntax for container recipes, but the function is the same.

Image

An image is a read-only template that contains all the code, dependencies, libraries, and supporting files that are required to launch a container. Docker stores images as layers, and any changes made to an image are captured by adding new layers. The “base image” is the bottom-most layer that does not depend on any other layer and typically defines the operating system for the container.

Container

A container is an instance of an image that can execute a software environment. Running a container requires a container runtime environment (e.g. Docker, Apptainer) and an instruction set architecture (e.g. x86) compatible with the image from which the container is instantiated.

Image Registry

Docker images can be stored in online image registries, such as [Docker Hub](#). (It is analogous to the way Git repositories are stored on GitHub.) Image registries are an excellent way to publish research software and to discover tools built by others. Image registries support the notion of tags to identify specific versions of images.

Image Tags

Docker supports image tags, similar to tags in a git repository. Tags identify a specific version of an image. The full name of an image on Docker Hub is comprised of components separated by slashes. The components include an “owner” (which could be an individual or organization), the “name”, and the “tag”. For example, an image with the full name

```
tacc/gateways19:0.1
```

would reference the “gateways19” image owned by the “tacc” organization with a tag of “0.1”.

1.1.5 Summing Up

If you are developing an app or web service, you will almost certainly want to work with containers. First you must either **build** an image from a Dockerfile, or **pull** an image from a public registry. Then, you can **run** (or deploy) an instance of your image as a container.

1.2 Getting Started With Docker

1.2.1 Prerequisites

1) 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

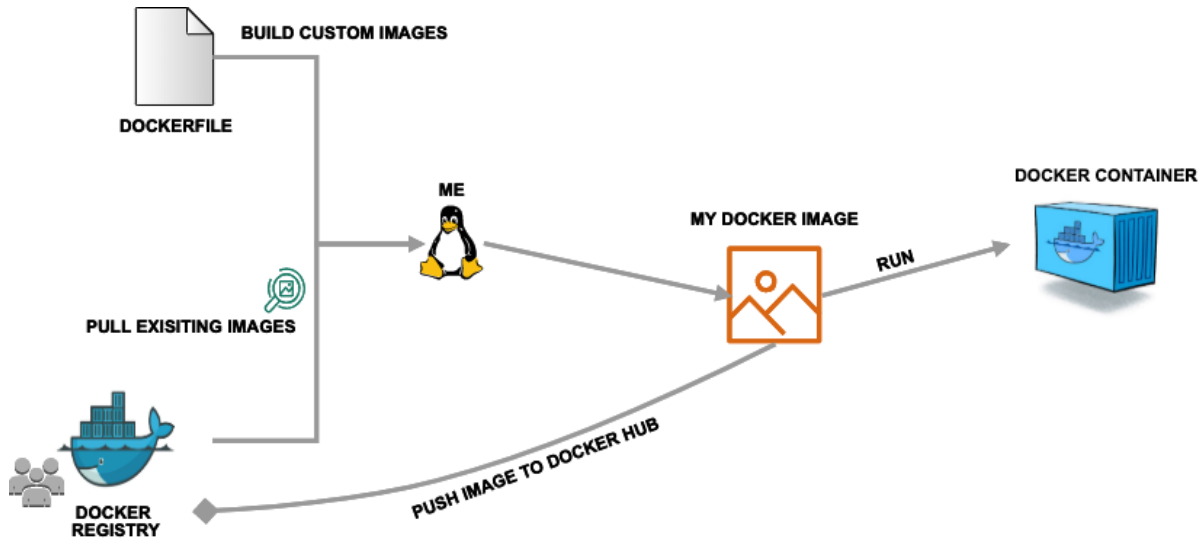


Fig. 3: Simple Docker workflow.

```

$ docker version
Client: Docker Engine - Community
 Version:           20.10.12
 API version:       1.41
 Go version:        go1.16.12
 Git commit:        e91ed57
 Built:             Mon Dec 13 11:45:41 2021
 OS/Arch:           linux/amd64
 Context:           default
 Experimental:      true

Server: Docker Engine - Community
 Engine:
  Version:           20.10.12
  API version:       1.41 (minimum version 1.12)
  Go version:        go1.16.12
  Git commit:        459d0df
  Built:             Mon Dec 13 11:44:05 2021
  OS/Arch:           linux/amd64
  Experimental:      false
 containerd:
  Version:           1.4.12
  GitCommit:        7b11cfaabd73bb80907dd23182b9347b4245eb5d
 runc:
  Version:           1.0.2
  GitCommit:        v1.0.2-0-g52b36a2
 docker-init:
  Version:           0.19.0
  GitCommit:        de40ad0

```

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

- 2) Create a [Docker Hub](#) account

Having a Docker Hub account makes it easier to share your containers with other researchers. Use the Docker CLI to login to Docker Hub to be able to push images to your repository:

```
$ docker login
(Enter username/password)
```

3) Create a TACC Account

We will use the TACC account later in this workshop to log in to Frontera and run our containers on an HPC system with Singularity (Apptainer).

EXERCISE

While everyone gets set up, take a few minutes to run `docker --help` and a few examples of `docker <verb> --help` to make sure you can find and read the help text.

1.2.2 Working with Images from Docker Hub

To introduce ourselves to some of the most essential Docker commands, we will go through the process of listing images that are currently available on our local machines, and we will pull a ‘hello-world’ image from Docker Hub. Then we will run the ‘hello-world’ image to see what happens.

List images on your local machine with the `docker images` command. This peaks into the Docker daemon, which is shared by all users on this system, to see which images are available, when they were created, and how large they are:

```
$ docker images
REPOSITORY          TAG          IMAGE ID          CREATED          SIZE
ubuntu              18.04       6ad7e71ba7d     2 days ago     63.2MB
```

Note: If this is your first time using Docker, you may not have any images stored on your local machine.

Pull an image from Docker hub with the `docker pull` command. This looks through the Docker Hub registry and downloads the ‘latest’ version of that image:

```
$ docker pull hello-world
Using default tag: latest
latest: Pulling from library/hello-world
2db29710123e: Pull complete
Digest: sha256:10d7d58d5ebd2a652f4d93fdd86da8f265f5318c6a73cc5b6a9798ff6d2b2e67
Status: Downloaded newer image for hello-world:latest
docker.io/library/hello-world:latest
```

Run the image we just pulled with the `docker run` command. In this case, running the container will execute a simple shell script inside the container that has been configured as the ‘default command’ when the image was built:

```
$ docker run hello-world

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.
```

(continues on next page)

(continued from previous page)

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.

To try something more ambitious, you can run an Ubuntu container with:

```
$ docker run -it ubuntu bash
```

Share images, automate workflows, and more with a free Docker ID:

<https://hub.docker.com/>

For more examples and ideas, visit:

<https://docs.docker.com/get-started/>

Verify that the image you just pulled is now available on your local machine:

```
$ docker images
REPOSITORY          TAG          IMAGE ID          CREATED          SIZE
ubuntu              18.04       6ad7e71ba7d     2 days ago     63.2MB
hello-world        latest      feb5d9fea6a5    7 months ago   13.3kB
```

Check to see if any containers are still running using `docker ps`:

```
$ docker ps
CONTAINER ID   IMAGE          COMMAND          CREATED          STATUS          PORTS          NAMES
```

EXERCISE

The command `docker ps` shows only currently running containers. Pull up the help text for that command and figure out how to show all containers, not just currently running containers.

EXERCISE

Closely inspect metadata for your downloaded image(s) using the `docker inspect` command:

```
$ docker inspect hello-world
[
  {
    "Id": "sha256:feb5d9fea6a5e9606aa995e879d862b825965ba48de054caab5ef356dc6b3412
→",
    "RepoTags": [
      "hello-world:latest"
    ]
    ...
  }
]
```

1.2.3 Docker Core Commands

Command	Usage
docker login	Authenticate to Docker Hub using username and password
docker images	List images on the local machine
docker ps	List containers on the local machine
docker pull	Download an image from Docker Hub
docker run	Run an instance of an image (a container)
docker inspect	Provide detailed information on Docker objects
docker rmi	Delete an image
docker rm	Delete a container
docker stop	Stop a container
docker build	Build a docker image from a Dockerfile in the current working directory
docker tag	Add a new tag to an image
docker push	Upload an image to Docker Hub

1.2.4 Additional Resources

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
```

- [Official Docker Documentation](#)

1.3 Working with Docker

In this section, we will look at a few more advanced examples of working with container images that are available on [Docker Hub](#). We will run commands using the containerized assets both *interactively* and *non-interactively*.

1.3.1 Pull An Official Image

One powerful aspect of developing with containers and the Docker ecosystem is the large collection of container images freely available. There are 10s of millions of images on Docker Hub alone, but beware: using an image that you don't know anything about comes with the same risks involved with running any software.

Warning: Be careful running container images that you are not familiar with. Some could contain security vulnerabilities or, even worse, malicious code like viruses or ransomware.

To combat this, Docker Hub provides “[Official Images](#)”, a well-maintained set of container images providing high-quality installations of operating systems, programming language environments and more.

We can search through the official images on Docker Hub [here](#).

Scroll down to find the Python official image called `python`, then click on that [image](#).

We see a lot of information about how to use the image, including information about the different “tags” available. We see tags such as 3.12-rc, 3.11.2, 3.11, 3, etc. We’ll discuss tags in detail later, but for now, does anyone have a guess as to what the Python tags refer to?

We can pull the official Python image using command, then check to make sure it is available locally:

```
$ docker pull python
...
$ docker images
...
$ docker inspect python
...
```

Tip: Use `docker inspect` to find some metadata available for each image.

1.3.2 Start an Interactive Shell Inside a Container

Using an interactive shell is a great way to poke around inside a container and see what is in there. Imagine you are ssh-ing to a different Linux server, have root access, and can see what files, commands, environment, etc., is available.

Before starting an interactive shell inside the container, execute the following commands on your local device (we will see why in a minute):

```
$ whoami
wallen
```

```
$ pwd
/Users/wallen
```

```
$ uname -a
Darwin wireless-10-147-13-120.public.utexas.edu 22.1.0 Darwin Kernel Version 22.1.0:
↪Sun Oct 9 20:14:54 PDT 2022; root:xnu-8792.41.9~2/RELEASE_X86_64 x86_64
```

Now start the interactive shell inside a Python container:

```
$ docker run --rm -it python /bin/bash
root@fc5b620c5a88:/#
```

Here is an explanation of the command options:

```
docker run      # run a container
--rm            # remove the container when we exit
-it            # interactively attach terminal to inside of container
python         # use the official python image
/bin/bash      # execute the bash shell program inside container
```

Try the following commands - the same commands you did above before starting the interactive shell in the container - and note what has changed:

```
root@fc5b620c5a88:/# whoami
root
```

```
root@fc5b620c5a88:/# pwd
/
```

```
root@fc5b620c5a88:/# uname -a
Linux e8bdfbf0fad7 5.10.104-linuxkit #1 SMP Thu Mar 17 17:08:06 UTC 2022 x86_64 GNU/
↪Linux
```

Now you are the `root` user on a different operating system inside a running Linux container! You can type `exit` to escape the container.

EXERCISE

Before you exit the container, try running the command `python`. What happens? Compare that with running the command `python` directly on your local device.

1.3.3 Run a Command Inside a Container

Back out on your local device, we now know we have a container image called `python` that has a particular version of Python (3.11.x) that may not otherwise be available on your local device. The 3.11.x Python interpreter, its standard library, and all of the dependencies of those are included in the container image and are *isolated* from everything else. This image (`python`) is portable and will run the exact same way on any OS that Docker supports.

In practice, though, we don't want to start interactive shells each time we need to use a software application inside an image. Docker allows you to spin up an *ad hoc* container to run applications from outside. For example, try:

```
$ docker run --rm python whoami
root
```

```
$ docker run --rm python pwd
/
```

```
$ docker run --rm python uname -a
Linux e8bdfbf0fad7 5.10.104-linuxkit #1 SMP Thu Mar 17 17:08:06 UTC 2022 x86_64 GNU/
↪Linux
```

```
$ docker run -it --rm python
Python 3.11.2 (main, Feb 11 2023, 02:24:27) [GCC 10.2.1 20210110] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> # type exit() to escape
```

The first three commands above omitted the `-it` flags because they did not require an interactive terminal to run. On each of these commands, Docker finds the image the command refers to, spins up a new container based on that image, executes the given command inside, prints the result, and exits and removes the container.

The last command, which did not specify a command to run inside the container, uses the container's default command. We don't know ahead of time what (if any) default command is provided for any given image, but what default command was provided for the `python` image?

Yes, it was the `python` command itself, and that requires an interactivity to use, so we provide the `-it` flags.

If all else fails, display the help text:

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

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


1.3.4 Additional Resources

- [Docker Docs](#)
- [Best practices for writing Dockerfiles](#)
- [Docker Hub](#)
- [Docker for Beginners](#)
- [Play with Docker](#)

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, on an HPC cluster, or in the cloud.

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/
```

(continues on next page)

(continued from previous page)

```
$ ls
Dockerfile
```

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

```
1  #!/usr/bin/env python3
2  from random import random as r
3  from math import pow as p
4  from sys import argv
5
6  # Make sure number of attempts is given on command line
7  assert len(argv) == 2
8  attempts = int(argv[1])
9  inside = 0
10 tries = 0
11
12 # Try the specified number of random points
13 while (tries < attempts):
14     tries += 1
15     if (p(r(),2) + p(r(),2) < 1):
16         inside += 1
17
18 # Compute and print a final ratio
19 print( f'Final pi estimate from {attempts} attempts = {4.*(inside/tries)}' )
```

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
ubuntu: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 or other namespace where you have permissions to publish. 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:

```
$ 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 git@github.com:username/pi-estimator.git
$ git branch -M main
$ git push -u origin main
```

Note: This assumes you have previously added an [SSH key to your GitHub account](#) for the machine you are working on.

Make sure to use the GitHub URI 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
```

DockerHub also has integrations to automatically update your image in the public container registry every time you commit new code, but they are no longer available unless you have a paid account.

For example, see: [Set up automated builds](#)

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:

```
1  #!/usr/bin/env python3
2  from random import random as r
3  from math import pow as p
4  from sys import argv
5
6  # Use argparse to take command line options and generate help text
7  import argparse
8  parser = argparse.ArgumentParser()
9  parser.add_argument("number", help="number of random points (int)", type=int)
10 args = parser.parse_args()
11
12 # Grab number of attempts from command line
13 attempts = args.number
14 inside = 0
15 tries = 0
16
17 # Try the specified number of random points
18 while (tries < attempts):
19     tries += 1
```

(continues on next page)

(continued from previous page)

```

20     if (p(r(),2) + p(r(),2) < 1):
21         inside += 1
22
23     # Compute and print a final ratio
24     print( f'Final pi estimate from {attempts} attempts = {4*(inside/tries)}' )

```

(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`:

```

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 +x /code/pi.py
8
9  ENV PATH "/code:$PATH"
10
11 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  # 1: edit pi.py to include new code
2  # 2: edit Dockerfile to include CMD instruction
3  # 3: follow the steps below
4
5  $ docker build -t username/pi-estimator:0.2 .
6  $ docker run --rm username/pi-estimator:0.2
7  usage: pi.py [-h] number
8
9  positional arguments:
10     number          number of random points (int)
11
12  optional arguments:
13     -h, --help      show this help message and exit
14
15  $ docker run --rm username/pi-estimator:0.2 pi.py 1000000
16  Final pi estimate from 1000000 attempts = 3.143672
17
18  $ docker run --rm -it username/pi-estimator:0.2 /bin/bash
19  root@c5aa145e5546:/# which pi.py
20  /code/pi.py
21  root@c5aa145e5546:/# pi.py 1000000
22  Final pi estimate from 1000000 attempts = 3.141168
23  root@c5aa145e5546:/# exit

```

(continues on next page)

(continued from previous page)

```
24  
25 $ docker push username/pi-estimator:0.2
```

2.3.4 Commit to GitHub (Again)

Finally, push your modified code to GitHub and tag the release as 0.2:

```
$ 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.

2.3.5 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

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 (now *Apptainer*) 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/Apptainer

Note:

Prerequisites This section uses the Frontera compute cluster to run Singularity. An active allocation on Frontera 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.

Note: Singularity has rebranded itself as Apptainer, and they are the same software. At some point, the name of the executable will change as well, but this has not happened yet on all systems at TACC. For now, we will stick to calling

it Singularity to avoid confusion. More info is here: <https://apptainer.org/>

3.1.1 Login to Frontera

For today's training, we will use the Frontera supercomputer, the 19th 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 Frontera system. Instructions depend on your laptop's operating system.

Mac / Linux:

```
Open the application 'Terminal'  
ssh username@frontera.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: frontera.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
```

(continues on next page)

(continued from previous page)

```
$ 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 inadvertently 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 Frontera compute cluster to run Singularity. An active allocation on Frontera 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 --version
```

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 rtx                 # Queue name
#SBATCH -N 1                   # Total number of nodes requested (56_
↳cores/node)
#SBATCH -n 1                   # Total number of mpi tasks requested
#SBATCH -t 02:00:00           # Run time (hh:mm:ss) - 4 hours
#SBATCH --reservation Containers_Class_RTX # a reservation only active during the_
↳training

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

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 Frontera system, the user guide is at: <https://frontera-portal.tacc.utexas.edu/user-guide/>

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 Frontera

If you are not already, please login to the Frontera 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 Frontera, 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
→to jobId)
#SBATCH -p rtx                  # Queue name
```

(continues on next page)

(continued from previous page)

```

#SBATCH -N 1 # Total number of nodes requested (56_
↪cores/node)
#SBATCH -n 1 # Total number of mpi tasks requested
#SBATCH -t 00:10:00 # Run time (hh:mm:ss)
#SBATCH --reservation Containers_Class_RTX # a reservation only active during the_
↪training

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 "wjallen" 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 Frontera, 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	Lonestar6	Local Dev
tacc/tacc-centos7-mvapich2.3-ib	✓		✓	✓	✓
tacc/tacc-centos7-mvapich2.3-psm2		✓			
tacc/tacc-centos7-impi19.0.7-common	✓	✓		✓	✓
tacc/tacc-ubuntu18-mvapich2.3-ib	✓		✓	✓	✓
tacc/tacc-ubuntu18-mvapich2.3-psm2		✓			
tacc/tacc-ubuntu18-impi19.0.7-common	✓	✓		✓	✓

Note: The singularity version of these containers should be invoked with `singularity run on HPC`.

In this tutorial, we will be using the `tacc/tacc-ubuntu18-impi19.0.7-common` image to satisfy the MPI architecture on 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 Frontera.

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 Frontera.

```
$ 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-impil9.0.7-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 Frontera

To start, lets allocate a single [GPU Node](#), which has 16 physical Intel cores and 4 NVIDIA Quadro RTX 5000 GPUs 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 rtx -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 do not go over your $HOME quota
$ cd $SCRATCH

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

Run container sequentially
$ 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 rtx -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 rtx                       # Queue name
#SBATCH -N 1                         # Total number of nodes requested (56
↳cores/node)
```

(continues on next page)

(continued from previous page)

```
#SBATCH -n 8 # Total number of mpi tasks requested
#SBATCH -t 00:10:00 # Run time (hh:mm:ss)
#SBATCH --reservation Containers_Class_RTX # a reservation only active during the_
↳training

module load tacc-singularity
cd $SCRATCH
ibrun singularity run 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 find 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.

Base Docker images

When running at TACC, we have a set of curated Docker images that include [TensorFlow](#) and [PyTorch](#) for use in the FROM line of your own containers. You can see a list of available images at [Docker Hub](#) and the source at <https://github.com/TACC/tacc-ml>.

Image	Frontera/rtx	Maverick2	Lonestar6
tacc/tacc-ml:centos7-cuda9-tf1.14-pt1.3		✓	
tacc/tacc-ml:centos7-cuda10-tf1.15-pt1.3	✓	✓	
tacc/tacc-ml:centos7-cuda10-tf2.4-pt1.7	✓	✓	
tacc/tacc-ml:centos7-cuda11-tf2.6-pt1.10	✓	✓	✓
tacc/tacc-ml:ubuntu16.04-cuda9-tf1.14-pt1.3		✓	
tacc/tacc-ml:ubuntu16.04-cuda10-tf1.15-pt1.3	✓	✓	
tacc/tacc-ml:ubuntu16.04-cuda10-tf2.4-pt1.7	✓	✓	
tacc/tacc-ml:ubuntu20.04-cuda11-tf2.6-pt1.10	✓	✓	✓

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

Building a GPU aware container

In the previous two examples, we have used pre-built containers to test GPU capability. Here we are going to build a GPU aware container to do some NLP/Text classification with the [BERT](#) transformer model using PyTorch. We are going to use one of the TACC base images (`tacc/tacc-ml:centos7-cuda10-tf2.4-pt1.7`) as a starting point.

On your local laptop, create a directory to build the “bert-classifier” Docker image and download (or copy and paste) the following files:

- [Dockerfile](#)
- [bert_classifier.py](#)
- [train.csv](#)

- test.csv
- valid.csv

Note: For speed, you can also clone the repository from <https://github.com/eriksf/bert-classifier.git>.

Take a look at the files. `bert_classifier.py` is a Python script that uses PyTorch to do the text classification. The Dockerfile is based on the **tacc/tacc-ml:centos7-cuda10-tf2.4-pt1.7** base image and installs a couple of needed python libraries in addition to moving the datasets into the image. `train.csv`, `test.csv`, and `valid.csv` are pre-processed CSV files containing the training, test, and validation datasets.

With these files downloaded, we can now build the image.

```
$ docker build -t USERNAME/bert-classifier:0.0.1 .
```

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

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 Frontera.

```
$ docker push USERNAME/bert-classifier:0.0.1
```

Testing the Container Locally with CPU

As was mentioned above, before using TACC allocation hours, it's a good idea to test locally. In this case, we can at least test that the program help works.

```
$ docker run --rm -it eriksf/bert-classifier:0.0.1 bert_classifier.py -h
usage: bert_classifier.py [-h] [-d DEVICE] [-o OUTPUT] [-s SOURCE]

optional arguments:
  -h, --help            show this help message and exit
  -d DEVICE, --device DEVICE
                        The device to run on: cpu or cuda (DEFAULT: cuda)
  -o OUTPUT, --output OUTPUT
                        The output folder (DEFAULT: current directory)
  -s SOURCE, --source SOURCE
                        The location of the data files (DEFAULT: /code in
                        container)
```

We could even test the classification (very slowly) using the CPU.

```
$ docker run --rm -it eriksf/bert-classifier:0.0.1 bert_classifier.py -d cpu
```

Running the Container on Frontera

To start, lets allocate a single RTX node, which has 4 NVIDIA Quadro RTX 5000 GPUs with 16 GB of Memory each.

```
$ idev -m 60 -p rtx
```

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

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

Change to $SCRATCH directory
$ cd $SCRATCH

Pull container
$ singularity pull docker://USERNAME/bert-classifier:0.0.1

Run container
$ singularity exec --nv bert-classifier_0.0.1.sif bert_classifier.py
```

CHAPTER 4

Indices and tables

- `genindex`
- `modindex`
- `search`