Singularity: A Container System for HPC Applications

Dennis Gannon, Vanessa Sochat
(This is a supplementary chapter to the MIT Press book “Cloud Computing for Science Engineering” [6])

In Chapter 6 we described the basic idea behind containerized applications and discussed Docker in some detail. Docker allows us to build an application container that will run on any machine that also has Docker, and we don’t need to make any changes to the machine itself. This is a fantastic way to share ready-to-run applications such as the tutorial container we built for this book, a Jupyter notebook server, or a scientific application. In chapter 7 we discussed the various ways to scale applications in the cloud to exploit more parallelism. We discussed how Docker containers can be used as microservices that run on AWS, Azure and Google in their cloud-scale container management systems that are based on platforms like Kubernetes and Mesos. In Chapter 7.2 we also described how you can build an HPC style cluster in the cloud to run MPI programs, however you might have noticed that we did not discuss ways to run Docker-based MPI programs on a distributed cluster. The reason for this is because it is not efficient to do with Docker. In fact, while Docker is prime for many applications, it doesn’t satisfy the functional and security needs that are common in high performance computing (HPC) supercomputer environments.

Scientists, academics, and the larger community reliant on these shared resources were waiting for Docker to support the HPC community, but administrators of these clusters simply could not install it. They had good reason. A supercomputer is a shared resource where having root privileges comes down to an ability to act in a malicious way to the files and folders of any other user. In that the Docker daemon could allow a user to escalate to root, it was a no starter.

However, it was clear that the need for reproducible, container-based environments was especially important for science, and in fact, would help both users and cluster administrators. Given a wide range of scientific communities, each with varying needs, centers are typically forced to provide standard libraries that address the needs of most users, but neglect the long tails of science. With containers, even the most niche software could be installed, and further, the user is empowered to do this. What was needed by the HPC system managers was a secure way to allow containerization that would also allow access to the advanced networking and other hardware on their systems.

It was finally in 2015 that the open source community, and specifically, the open source HPC community, responded. Jacobsen and Cannon from NERSC addressed this need with Shifter [1]. Another version of HPC center containerization was developed at Los Alamos National Labs by Priedhorsky and Randles and is called CharlieCloud [2]. Gregory M. Kurtzer from Lawrence Berkeley Lab developed [3,4] the linux container that is the topic of this chapter, as it may be the most widely used of these HPC container systems.

A Gentle Introduction to Singularity.

Singularity containers differ from Docker containers in several important ways, including the handling of namespaces, user privileges, and the images themselves. For a detailed technical description of these topics, read the CharlieCloud paper cited above. Our discussion here is intended to give you the basic ideas.

Permissions

With Docker, the process inside the container is running with "root" privileges in the container's OS and with the container's filesystem. If you are running the Docker container on a VM or other system where you have root privilege (such as you own PC) this is not a big deal. However, if you are running on a shared, bare-metal cluster such as a university or lab supercomputer this is a very serious problem. If a user is able to interact with the Docker daemon, it would be possible to obtain root status and potentially act maliciously. Singularity solves this by creating an image that runs with the user's identity and not root. The permissions that the user has inside the container are the same permissions as outside the container, and thus the security concern is lifted.

Filesystem

Recall that when Docker launches an image it is run as a process on the host that has no direct access to the host's filesystem except for those directories that are explicitly cross mounted into the container. There are pros and cons to a completely isolated environment, but for scientific computing, it's usually the case that interaction is needed with files, data, and libraries on the host. Here we come to another fundamental difference - Singularity does not completely close off the container's filesystem. In fact, the process that is running as the user sees the user's home directory, and the user's environment is shared seamlessly. Networking stacks such as TCP and those needed by MPI that are *not* exposed in a Docker container *are* exposed inside the Singularity container. If you are familiar with Docker, you are probably used to mapping ports from the host to the container. This is because the network stack is private to the container, and unfortunately it makes Docker containers not suitable for more complicated networks like Infiniband. This is why, if you have ever tried executing MPI programs with Docker that use multiple container instances as workers, you will notice they are extremely awkward and slow. This goes against the entire point of using MPI in the first place, which is to support ultra-fast, synchronized communication.

Container Image

Have you ever wondered where on your computer your Docker images live? A fundamental difference, and perhaps the largest, between Singularity and Docker containers is the image format itself. Singularity is an actual file that you could put in entirety on your desktop, and move around as you would any file. Docker images come in layers, and they are assembled like layers of a cake to generate a running image. While it may seem optimal for download layers instead of a single image file, it introduces greater challenges for long term reproducibility of applications if they are dependent on many tiny pieces. To help solve this problem, both containers have developed registries for archive and sharing of containers, or pieces that turn into them.

The Singularity Hub

One of the great things about Docker is the Docker Hub, which is the place where containers can be shared. Singularity has its own Hub, but it works a little differently from Docker's. With the Docker Hub, you build and test your container on your own machine and then upload it to the Hub. You can then pull the container image from any machine that can contact the Docker Hub. The Singularity Hub actually builds the container from a build specification file, a file named "Singularity" that contains the steps needed to generate your container. The workflow is comfortable for the modern scientist that knows how to use Github. He or she commits the file to a Github repository, connects to Singularity Hub via https://www.singularity-hub.org, and then the container is built automatically on every push. The user has the ability to customize the builder, or the version of Singularity. A complete log is provided for debugging, and additional branches in the Github repository can be activated to build different tags for an image. By way of Github, each build specification file and the image it derives are associated with a version, the Github commit identifier. This is generally good practice for reproducibility of these applications.

Looking at the public images in the Singularity Hub you will see some very interesting examples. There are several versions of the deep learning Tensorflow system that take advantage of the fact that Singularity containers can see special hardware like GPUs that are not yet available inside Docker containers. Quantum state diffusion tools top the list of most frequent builds. The neuroscience software package NeuroDebian has also been "Singularity-ized" and there are a host of life science related pipelines. We can look at the "Compare All" tool to see a tree that compares all container collections on Singularity Hub, updated on a nightly basis. Importantly, any image built on Singularity Hub is immediately available by way of the Singularity software itself. This means that with a few Github commits and using Singularity with reference to the image on Singularity Hub, you can design, build, and use containers without having root privileges at all.

Final Thoughts

Container technology has revolutionized large scale cloud computing, and is now taking the HPC world by storm as well. It is now possible to build an application on your desktop and run hundreds or even thousands of instances, without change, on any of the public clouds. Supercomputing has gone through two "portability" revolutions. The first step was the nearly uniform adoption of Unix. While versions of Unix varied greatly, it made it easier for users to move from one system to another. The second revolution in portability was the introduction and wide adoption of MPI. Adopting Singularity or one of the other HPC container technologies can usher in a third revolution in HPC application portability. The last missing piece of technology is a way to share containers. We have started to see solutions emerge such as BioContainers, different technologies centered around Github, and as we discussed, the hubs provided by Docker and Singularity.

Finally, it is notable that Singularity has grown out of the open source movement. The core team of developers is a mixed group of HPC architects, research software engineers, and academics. Singularity Hub was and is still maintained by a single individual. This general observation is a motivating in that it teaches us that great ideas can come from unexpected places, and sometimes it's worth building the tool or software that you want and need instead of waiting for a larger entity to provide it.

A Small Tutorial

In the following paragraphs we illustrate Singularity with a simple example intended to show how to contain an MPI program and how to use Singularity Hub to share the image.

To create a Singularity container you must first install Singularity. We chose to do this on an instance of the AZURE data science VM on an Azure machine with 16 cores. This is a very useful VM because of all the excellent pre-installed libraries. It is running CentOS Linux Version 7. Installing Singularity was easy.

```
$VERSION=2.3
$wget https://github.com/singularityware/singularity/
     releases/download/$VERSION/singularity-$VERSION.tar.gz
$tar xvf singularity-$VERSION.tar.gz
$cd singularity-$VERSION
$./configure --prefix=/usr/local
$make
$sudo make install
```

The next step is to create a Singularity container image. You first create a basic container image with the command

```
$singularity create --size 2048 mysing-container.img
```

We specified the size to be 2Gig to be big enough to load lots of data and libraries, but the default size of around 800Meg is big enough for what we show here.

Next we need to load an OS and our application libraries and binaries. Because both Singularity and CharlieCloud are based on the same Linux capabilities as Docker it is possible to load the contents of a Docker container file system into Singularity and CharlieCloud containers. In the case of Singularity you can also load items from the command line one at a time, but it is better to bootstrap the container from a Singularity definition specification file (which we will refer to below as a Singularity "spec" file). The one we used is shown on the next page and explained below. This file is called "Singularity”.

```
BootStrap: docker
From:dbgannon/ubuntuplus

%files

%labels
MAINTAINER gannon

%environment
LD_LIBRARY_PATH=/usr/local/lib
export LD_LIBRARY_PATH=/usr/local/lib

%runscript
   echo "This is what happens when you run the container..."
   /usr/bin/ring-simple

%post
   echo "Hello from inside the container"
   apt-get -y update
   apt-get -y upgrade
   apt-get -y install vim
   apt-get -y install build-essential
   apt-get -y install wget
   wget https://www.open-mpi.org/software/ompi/v2.1/downloads/openmpi-2.1.0.tar.bz2
   tar -xf openmpi-2.1.0.tar.bz2
   echo "installing openmpi"
   ls
   cd openmpi-2.1.0
   ./configure --prefix=/usr/local
   make
   make install
   echo "done"
   export LD_LIBRARY_PATH=/usr/local/lib
   ls /mpicodes
   mpicc /mpicodes/ring-simple.c -o /usr/bin/ring-simple
```

There are six important parts to this spec file: a header and sections that are defined by % keywords. The header tells Singularity we are building the container from a docker image called "dbgannon/ubuntuplus". This docker file for dbgannon/ubuntuplus is very simple. It contains only the basic ubuntu OS and a directory containing our source codes called "mpicodes".

We are going to build the container on our VM (where we have sudo privileges) with the command

```
%sudo /usr/lib/bin/singularity bootstrap  mysing-container.img Singularity
```

where "Singularity" is the build specification file above. This bootstrap operation executes the script in the "post" section of the file. The post script updates the ubuntu OS, adds standard build tools and does a complete install of OpenMPI which puts the correct MPI libraries in /usr/local/lib in the container filespace. Finally, it compiles a simple mpi program called ring-simple and stores the executable in /usr/bin which is a standard location for executables that are shared. Note that we executed this with "root" privilege using sudo, so it is possible to modify this directory.

When the bootstrap completes the post script it executes the "files" section. This is empty here, but its role is to copy files from a host source path to a path in the container. It then set the environment variable in the manner described in the “environment” section.

Once the bootstrap is complete, we have several ways to launch the container. One is simply enters the command

```
$bash ./mysing-container.img
```

which runs the "runscript" as defined in the Singularity specification file. The other method is

```
$singularity exec mysing-container.img command
```

which executes the "command" in the container's OS with the user's identity. We illustrate both in the paragraphs that follow.

The MPI program ring-simple.c that we have included is shown below. It is basically half of a ring. mpi node 0 sends -1 to node 1. node 1 sends 0 to node 2, etc until the highest ranking node receives the message and the program exits. It is a trivial test that the MPI communications are working.

```
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int main(int argc, char** argv) {
 // Initialize the MPI environment
 MPI_Init(NULL, NULL);
 // Find out rank, size
 int world_rank;
 MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
 int world_size;
 MPI_Comm_size(MPI_COMM_WORLD, &world_size);
 char hostname[1024];
 gethostname(hostname, 1024);

 // We are assuming at least 2 processes for this task
 if (world_size < 2) {
   fprintf(stderr, "World size must be greater than 1 for %s\n", argv[0]);
   MPI_Abort(MPI_COMM_WORLD, 1);
 }
 //printf("world size =%d\n", world_size);
 int number;
 int i;
 if (world_rank == 0) {
   // If we are rank 0, set the number to -1 and send it to process 1
   printf("Process 0 sending -1 to process 1\n");
   number = -1;
   MPI_Send(&number, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
  } else if (world_rank > 0 && world_rank < world_size) {
   MPI_Recv(&number, 1, MPI_INT, world_rank-1, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
   printf("Process %d received number %d from process %d on node %s\n", world_rank, number,
world_rank-1, hostname);
   number = number+1;
   if (world_rank < world_size-1){
       MPI_Send(&number, 1, MPI_INT, world_rank+1, 0, MPI_COMM_WORLD);
   }
 }
 MPI_Finalize();
}
```

To run this MPI program we need to execute "mpirun", which was not installed on our Azure

VM. To install OpenMPI on the VM, we executed the same OpenMPI install as shown in the "post" script. Once that was done we executed the container just like a regular program with

```
$mpirun -np 8 mysing-container.img
```

This will invoke the "runscript" from the spec file.
The output is

```
This is what happens when you run the container...
This is what happens when you run the container...
This is what happens when you run the container...
This is what happens when you run the container...
This is what happens when you run the container...
This is what happens when you run the container...
This is what happens when you run the container...
This is what happens when you run the container...
Process 0 sending -1 to process 1
Process 1 received number -1 from process 0 on node myDSVM
Process 3 received number 1 from process 2 on node myDSVM
Process 4 received number 2 from process 3 on node myDSVM
Process 5 received number 3 from process 4 on node myDSVM
Process 2 received number 0 from process 1 on node myDSVM
Process 6 received number 4 from process 5 on node myDSVM
Process 7 received number 5 from process 6 on node myDSVM
```

Once we have our container there are a number of other Singularity commands we can use with it. One of the most useful is the "exec" command that allows you to execute commands in the container's OS. For example, we can open an interactive shell running in the container's OS with

```
$singularity exec mysing-container.img bash
```

```
$singularity exec mysing-container.img mpicc foobar.c -o foobar
$sudo /usr/local/bin/singularity copy foobar /usr/local/bin/foobar
```

Or we can compile another MPI program in the container's environment with the container's library and then move it to the container's /usr/local/bin using the singularity copy command with sudo as follows

Portability

To demonstrate that the container can be moved without change and executed on another system, we built an Amazon AWS CloudFormation Cluster consisting of 4 worker nodes each with 4 cores and a tiny headnode. These nodes were all running Amazon Linux AMIs (which we believe are Rhel fedora based). We followed the procedure outlined in the book in section 7.2.3. However, before we could get things to work, we needed to install Singularity on the head node AND on each of the worker nodes. Several things made this task simple. The AWS CloudFormation Cluster automatically mounts the head node user file space directly into the user file space of each worker node. Once we have completed the Singularity install on the head node, we need only log into each worker node and execute the commands

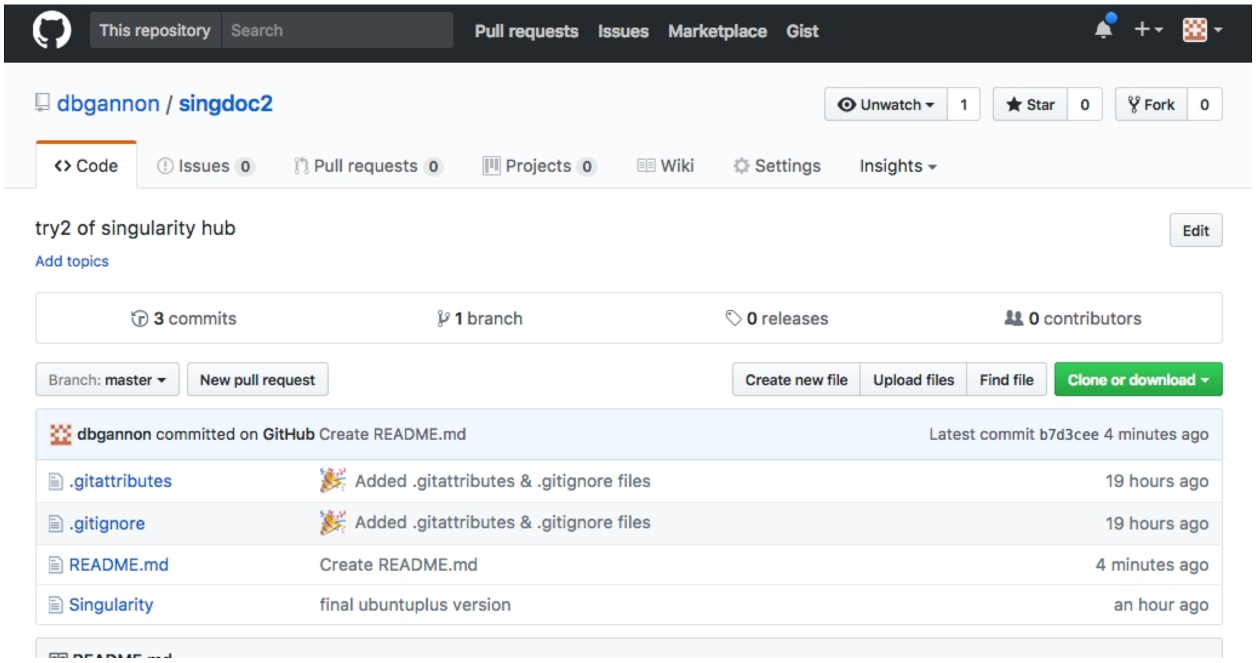
```
$cd singularity-2.3
$make install
```

This was easy to do with 4 nodes, but if we had 100 nodes we could submit a slurm job to do this.

We were able to build the Singularity container mysing-container.img exactly as before but it would not run with mpirun. The problem was that the AWS cluster installs the MPICH version of MPI and not openMPI. The MPICH mpirun was not compatible with the openmpi libraries in the container. This was easily resolved by deploying openmpi in the headnode and each worker and changing the default paths to point to openmpi. After this install everything worked as it should. We could do this because we completely own our virtual cluster in AWS. (The average user of a supercomputer center would need to convince the management to make this change if it was needed. We only know of a few centers that have taken this step.)

Using Singularity Hub

To use the Singularity Hub you put your Singularity spec file in a github repo as shown below. Here we have a github repository called dbgannon/singdoc2. It contains the Singularity spec file we created in the previous section.



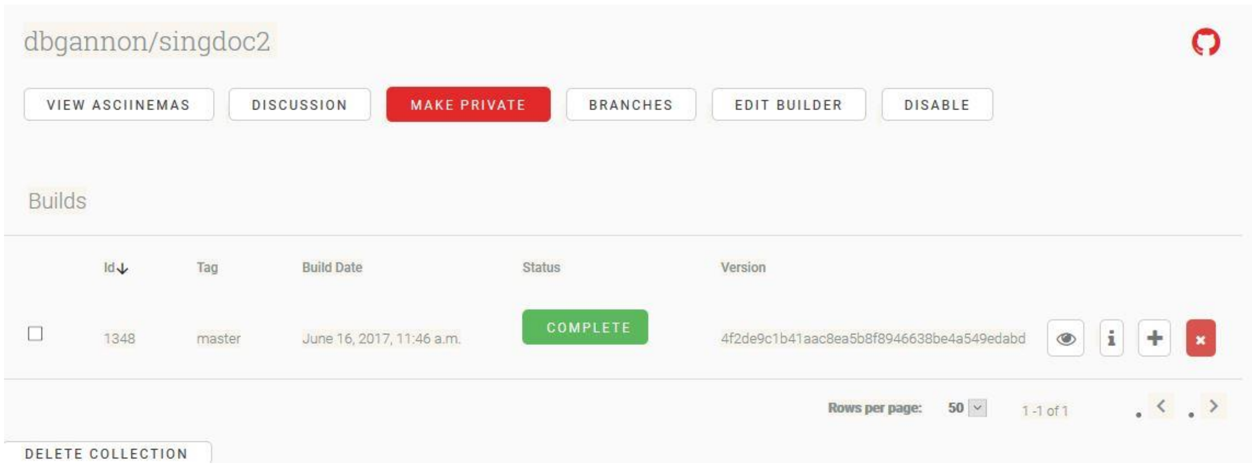
dbgannon / singdoc2

try2 of singularity hub

dbgannon committed on GitHub Create README.md — Latest commit b7d3cee 4 minutes ago

.gitattributes	Added .gitattributes & .gitignore files	19 hours ago
.gitignore	Added .gitattributes & .gitignore files	19 hours ago
README.md	Create README.md	4 minutes ago
Singularity	final ubuntuplus version	an hour ago

We next go to https://singularity-hub.org and login with our Github id. This asks your permission to import lots of things including all your repositories. Look for the repo with your Singularity file and select it. Singularity Hub will grab it and start to build it. When it is finished you will see something like the image below.


dbgannon/singdoc2

Builds

	Id↓	Tag	Build Date	Status	Version
☐	1348	master	June 16, 2017, 11:46 a.m.	COMPLETE	4f2de9c1b41aac8ea5b8f8946638be4a549edabd

Rows per page: 50 1 -1 of 1

Once the build is complete you can download it. We initially built and tested the container on the Azure datascience VM. To test the Singularity version we went to the AWS Cloudformation cluster and pulled the container from Singularity hub and ran it. The result is shown below. (Here the file “machines” contains the IP addresses of four 4-core worker nodes. As you can see, the mpirun distributed these evenly.)

```
$ singularity pull shub://dbgannon/singdoc2
Progress |===================================| 100.0%
Done. Container is at: ./dbgannon-singdoc2-master.img
$ mpirun -np 16 -machinefile machines singularity exec dbgannon-singdoc2-
master.img /usr/bin/ring-simple
Process 0 sending -1 to process 1
Process 1 received number -1 from process 0 on node ip-172-31-40-218
Process 2 received number 0 from process 1 on node ip-172-31-40-218
Process 3 received number 1 from process 2 on node ip-172-31-40-218
Process 4 received number 2 from process 3 on node ip-172-31-32-83
Process 5 received number 3 from process 4 on node ip-172-31-32-83
Process 6 received number 4 from process 5 on node ip-172-31-32-83
Process 7 received number 5 from process 6 on node ip-172-31-32-83
Process 8 received number 6 from process 7 on node ip-172-31-46-76
Process 9 received number 7 from process 8 on node ip-172-31-46-76
Process 10 received number 8 from process 9 on node ip-172-31-46-76
Process 11 received number 9 from process 10 on node ip-172-31-46-76
Process 12 received number 10 from process 11 on node ip-172-31-34-208
Process 13 received number 11 from process 12 on node ip-172-31-34-208
Process 14 received number 12 from process 13 on node ip-172-31-34-208
Process 15 received number 13 from process 14 on node ip-172-31-34-208
```

Works perfectly. No build required on the master node for the cluster. Just pull and run. The Singularity Hub site https://singularity-hub.org has good documentation that illustrates many other capabilities not discussed here. And an updated version is in the works.

A more interesting example.

The example above does not really illustrate the advantage of containerization very well. It gets more interesting when a complete application environment is containerized once so that it can be reused without change on multiple machines. To illustrate this we decided to show how Jupyter could be added to our container so that our MPI jobs could be launched interactively.

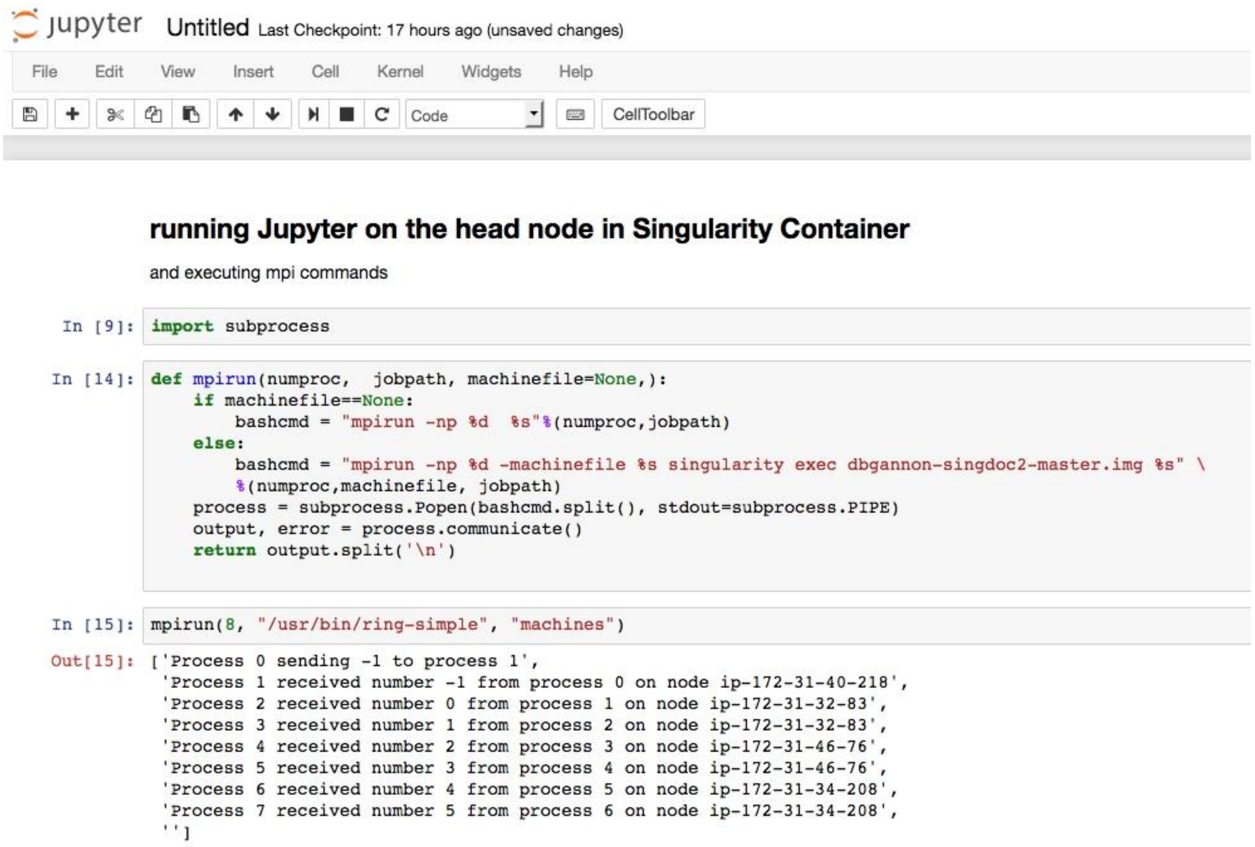
We built a Docker container from "continuumio/anaconda" that includes an up-to-date version of all the Python tools and Jupyter. We also included a directory that contained the necessary files to launch Jupyter along with our favorite mpi programs. The Singularity spec file was identical to the one described above with the exception of the addition of one more exported path.

```
export XDG_RUNTIME_DIR=/tmp/$UID
```

After testing this on our Azure datascience VM and pushing it to Singularity Hub we waited for the build to complete. Then running the commands on our AWS cluster

```
$ singularity pull shub://dbgannon/singdoc2
Progress |===================================| 100.0%
$ singularity exec dbgannon-singdoc2-master.img bash
$ bash /mpicodes/jupyterrun.sh
```

gives us a running jupyter instance. The image below shows a trivial Python notebook that can launch jobs on the cluster.



running Jupyter on the head node in Singularity Container

and executing mpi commands

```
In [9]: import subprocess

In [14]: def mpirun(numproc,  jobpath, machinefile=None,):
             if machinefile==None:
                 bashcmd = "mpirun -np %d  %s"%(numproc,jobpath)
             else:
                 bashcmd = "mpirun -np %d -machinefile %s singularity exec dbgannon-singdoc2-master.img %s" \
                 %(numproc,machinefile, jobpath)
             process = subprocess.Popen(bashcmd.split(), stdout=subprocess.PIPE)
             output, error = process.communicate()
             return output.split('\n')

In [15]: mpirun(8, "/usr/bin/ring-simple", "machines")

Out[15]: ['Process 0 sending -1 to process 1',
          'Process 1 received number -1 from process 0 on node ip-172-31-40-218',
          'Process 2 received number 0 from process 1 on node ip-172-31-32-83',
          'Process 3 received number 1 from process 2 on node ip-172-31-32-83',
          'Process 4 received number 2 from process 3 on node ip-172-31-46-76',
          'Process 5 received number 3 from process 4 on node ip-172-31-46-76',
          'Process 6 received number 4 from process 5 on node ip-172-31-34-208',
          'Process 7 received number 5 from process 6 on node ip-172-31-34-208',
          '']
```

The reader may notice that the way we execute mpirun when it is on the host machine (where we are running in the container's OS) differs from when we are doing remote executions (where we are running in the host OS). Another assumption we are making here is that we can open a port like 8887 on the head-node of the cluster to incoming traffic so we can run the Jupyter interface in our desktop browser. This is, of course, easy to do with our AWS cloudformation cluster, but likely not allowed on most institutional supercomputers.

Final Comments

We were interested in the performance overhead of using Singularity.   We made a simple set of measurements of a full ring program (similar to the one above but with a full loop of communication and run for 10000 cycles). The overhead of the singularity based version versus a "native" version was less than 2%. We attribute this to possible paging issues involving the size of the Singularity image (2GBytes) versus 2KBytes of the "native" object program.   For a serious application, we expect this overhead to vanish.

We hasten to add that Singularity is not just for MPI applications. Any application or pipeline that needs access to GPUs or other special hardware is going to want to use Singularity. In fact, it seems that the most common applications on Singularity Hub do not use MPI.

Bibliography

1. Jacobsen and Cannon, “Contain This, Unleashing Docker for HPC”, https://www.nersc.gov/assets/Uploads/cug2015udi.pdf, 2015
2. Priedhorsky and Randles, “ Charliecloud: Unprivileged containers for user-defined software stacks in HPC”, http://permalink.lanl.gov/object/tr?what=info:lanl-repo/lareport/LA-UR-16-22370 , 2017
3. Kurtzer, Sochat and Bauer, “Singularity: Scientific containers for mobility of compute”, PLOS1, 2017, http://journals.plos.org/plosone/article?id=10.1371/journal.pone.0177459
4. Singularity website http://singularity.lbl.gov/
5. Singularity Hub website   https://singularity-hub.org/
6. Foster and Gannon, “Cloud Computing for Science and Engineering”, MIT Press 2017. https://Cloud4SciEng.org.