Singularity: A Container System for MPI-based HPC Applications

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 without change on any machine the can run the docker engine. This is a fantastic way to share ready-to-run applications such as the tutorial container we built for this book, Jupyter and many others. 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 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 we did not discuss ways to run docker-based MPI programs on a distributed cluster. It can be done, but it is not very efficient and many of the advanced networking features of a supercomputer would not be visible inside a Docker container.

This proved to be a big problem for the people that run and manage HPC systems, because their users were demanding ways to bring the great ideas from containerization into their HPC environments. There are actually many reasons that the managers of HPC systems would like to have some sort of containerization mechanism available to their users. A big challenge for them is providing a computational environment that can address the needs of a wide variety of scientific communities where each community has a favorite flavor or Linux and OS-specific libraries and applications compiled with various compilers. HPC centers typically stick to one Linux flavor and standard libraries because to do more would be impossible to manage in a mulit-user environment. Unfortunately, these system managers did not trust Docker security for reasons we describe below. What was needed was a secure way to allow containerization and also the ability for the container to access the advanced networking and other hardware. Jacobsen and Cannon from NERSC addressed this with Shifter in https://www.nersc.gov/assets/Uploads/cug2015udi.pdf. Another version of HPC center containerization was developed at Los Alamos National Labs by Priedhorsky and Randles and is called CharlieCloud (see http://permalink.lanl.gov/object/tr?what=info:lanl-repo/lareport/LA-UR-16-22370 for an excellent discussion) Gregory M. Kurtzer from Lawrence Berkeley Lab developed Singularity (http://singularity.lbl.gov/) which may be the most widely used of these HPC container systems and we will discuss it here.

A Gentle Introduction to Singularity.

Singularity containers differ from Docker containers in several important ways. The major differences have to do with the way the two systems handle namespaces in Linux and the user privileges. To understand this in deep technical detail, read the CharlieCloud paper cited above. We discuss it here in a less technical level. First, recall that when Docker launches an image it is run as a process on the host that has no direct access to the hosts filesystem except for those directories that are explicitly cross mounted into the container. Furthermore 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. But if you are running on a shared, bare-metal cluster such as a university or lab supercomputer this is a very serious problem. Singularity creates an image that runs with the user's identity and not root. Furthermore Singularity does not completely close off the container's filesystem. The process, running as the user, sees the user's home directory on the host. In addition the dev namespace the Singularity process sees is the same one that the user normally sees on the host. This means that the TCP and other networking stacks that are available to the user are also available inside the singularity container. With a Docker container, the network stack is private to the container but there is a bridge to the host stack where you can map a port from one to the other. This make it possible for the docker container image to receive TCP messages, but fancy networks like Infiniband are not visible. And it makes the execution of MPI programs that use multiple container instances as workers extremely awkward and slow. The point of MPI is ultra-fast, synchronized communication.

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 used specified the size to be 2Gig to be big enough to load lost 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 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 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. The top part tells Singularity we are building the container from a docker image called "dbgannon/ubuntuplus". This docker file 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 spec 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. Finally it compiles a simple mpi program called ring-simple and stores the executable in /usr/bin. While this may seem like a strange location for our executables, it is not unreasonable. Note we executed this

with "root" privilege using sudo, so it is possible.

When the bootstrap completes the post script it executes the "file" spec. This is empty here, but it just copies files from a host source path to a path in the container. It then runs "environment" script.

We are now ready to launch and execute the container. The MPI program ring-simple.c that we have included is shown below. It is basically half 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. We executed the same OpenMPI install on the Azure VM 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 a shell running in the container's OS with

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

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

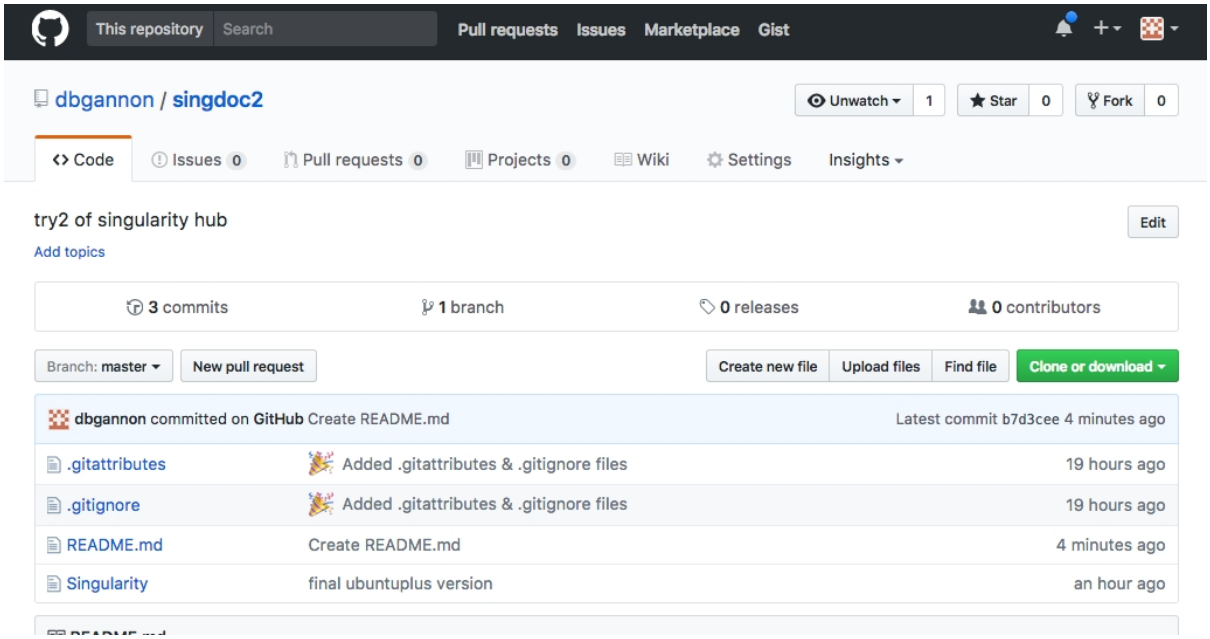
```
$singularity exec mysing-container.img mpicc foobar.c -o foobar
$sudo /usr/local/bin/singularity copy foobar /usr/local/bin/foobar
```

Portability

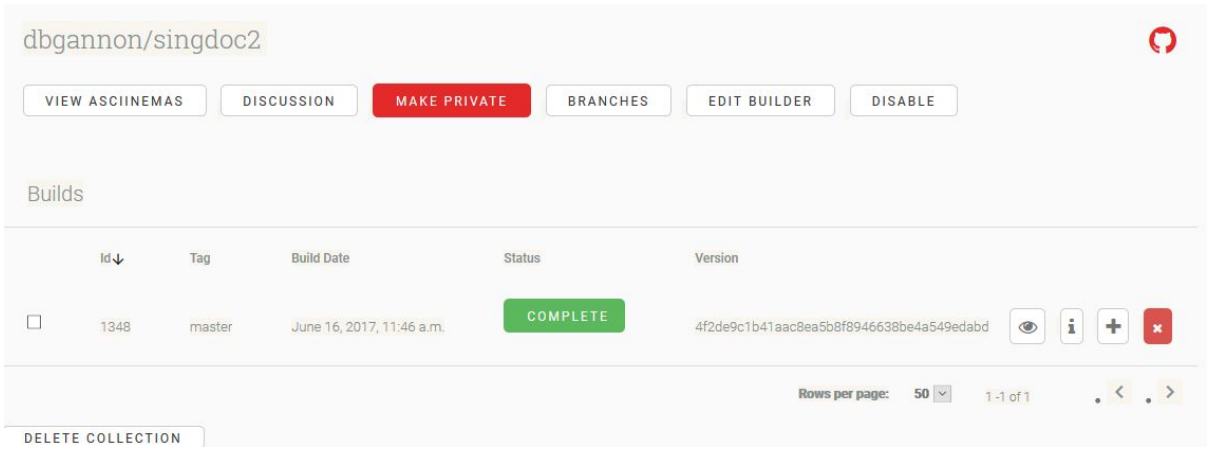
To demonstrate that the container can be moved without change and executed on another system, we built an Amazon AWS CloudFormation Cluster consisting of 4 worker nodes each with 4 cores and a tiny headnode. These nodes were all running Amazon Linux AMIs (which we believe are Rhel fedora based. We followed 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 thing made this task simple.

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 Hub. 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 con contact the Docker Hub. The Singularity Hub actually builds the container from the Singularity spec file for your container. 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.



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. This may take a while (We suspect it is being built on an overloaded server somewhere in the cloud or at Stanford.) When it is finished you will see something like the image below.



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 cloud formation cluster and pulled the container from Singularity hub and ran it. The result is shown below.

```
$ 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. WARNING: Singularity Hub cannot currently build any container. For best results use a Singularity file that builds from a Docker image such as the example here.

times for 16 process on 4 4-core aws cluster 22431.36328 22451.2793
21730.43359 21409.43359 22207.76758 21750.73633 21744.26367 21190.69531 21864.49658 basic

22827.90625 21936.76953 21991.34766 22588.66797 22378.5918
22442.50586 22137.10938 22245.96484 23035.63086 22398.27713 sing