Charliecloud 101

Friday, July 14, 2023 / 1:30 pm – 4:30 pm CST
Megan Phinney, Reid Priedhorsky

1 Getting started

1.1 Description

This workshop will provide participants with background and hands-on experience to use basic Charliecloud containers for HPC applications. We will discuss what containers are, why they matter for HPC, and how they work. We'll give an overview of Charliecloud, the unprivileged container solution from Los Alamos National Laboratory's HPC Division. Participants will build toy containers and a real HPC application, and then run them in parallel on a supercomputer. This will be a highly interactive workshop with lots of Q&A.

1.2 Prerequisites

1. Laptop or workstation with:
   - Portal access to ACES; and
   - `gitlab.com` account

Schedule (tentative)

1:15	*–*	*01:30*	*15*	*Teaching staff setup, last-minute Q&A*
01:30	–	01:50	20	Introduction to containers and Charliecloud (slides)
01:50	–	02:05	15	3. Key workflow operation: Pull
02:05	–	02:20	15	4. Containers are not special, part I: Alpine via tarball
02:20	–	02:35	15	5. Key workflow operation: Build from Dockerfile
02:35	*–*	*02:50*	*15*	*break & catch-up*
02:50	–	03:05	15	6. Key workflow operation: Push
03:05	–	03:35	30	7. MPI Hello World
03:35	–	03:50	15	8. Build Cache
03:50	–	04:30	40	ACES Examples

2 Pre-workshop setup checklist

2.1 Create working directory

We need to create a working directory. We will refer to it multiple times throughout the workshop so we will save the directory location in an environment variable.

```
$ export CHORKSHOP=~/chorkshop
$ mkdir $CHORKSHOP
```

3 Key workflow operation: Pull

To start, let’s obtain a container image that someone else has already built. The containery way to do this is the *pull* operation, which means to move an image from a remote repository into local storage of some kind.

First, let’s browse the Docker Hub repository of official AlmaLinux images.[1] Note the list of *tags*; this is a partial list of image versions that are available. We’ll use the tag “`8`”.

Use the Charliecloud program `ch-image` to pull this image to a directory.

> *Tip:* We strongly recommend you type out most of the commands in this manual, rather than copying and pasting, because then they will pass through your brain and you will learn more.

```
$ module load charliecloud
$ ch-image --help
usage: ch-image [-h] [--cache | --no-cache | --rebuild] [-a ARCH]
                [--always-download] [--auth] [--debug] [--dependencies]
                [--password-many] [-s DIR] [--tls-no-verify] [-v] [--version]
                CMD ...

Build and manage images; completely unprivileged.
[...]
$ ch-image pull --help
usage: ch-image pull [-h] [--cache | --no-cache | --rebuild] [-a ARCH]
                     [--always-download] [--auth] [--debug] [--dependencies]
                     [--password-many] [-s DIR] [--tls-no-verify] [-v]
                     [--version] [--last-layer N] [--parse-only]
                     IMAGE_REF [DEST_REF]
[...]
```

```
$ cd $CHORKSHOP
$ ch-image pull almalinux:8
pulling image:    almalinux:8
requesting arch:  amd64
manifest list: downloading: 100%
manifest: downloading: 100%
config: downloading: 100%
layer 1/1: 3239c63: downloading: 68.2/68.2 MiB (100%)
pulled image: adding to build cache
flattening image
layer 1/1: 3239c63: listing
validating tarball members
layer 1/1: 3239c63: changed 42 absolute symbolic and/or hard links to rela-
tive
resolving whiteouts
layer 1/1: 3239c63: extracting
```

[1] https://hub.docker.com/_/almalinux

```
image arch: amd64
done
$ ch-image list
almalinux:8
```

Images come in lots of different formats; `ch-run` can use directories and SquashFS archives. For this example, we'll use SquashFS. We use the command `ch-convert` to create a SquashFS image from `ch-image`'s internal storage directory, then run it.

Tip: Terminal activity in a container is written in blue.

```
$ ch-convert almalinux:8 almalinux.sqfs
$ ch-run almalinux.sqfs -- /bin/bash
$ pwd
/
$ ls
bin   ch   dev   etc   home   lib   lib64   media   mnt   opt   proc   root   run
sbin   srv   sys   tmp   usr   var
$ cat /etc/redhat-release
AlmaLinux release 8.8 (Sapphire Caracal)
$ exit
```

What does this command do?

1. Create a SquashFS-format image (`ch-convert ...`).
2. Create a container using that image (`ch-run almalinux.sqfs`).
3. Stop processing `ch-run` options (`--`). (This is standard notation for UNIX apps.)
4. Run the program `/bin/bash` inside the container, which starts an interactive shell where we enter a few commands and then exit, returning to the host.

4 Containers are not special, part I: Alpine via tarball

Many folks would like you to believe that containers are magic and special (especially if they want to sell you their container product). **This is not the case.** To demonstrate, we'll create a working container image using standard UNIX tools.

Many Linux distributions provide tarballs containing installed base images, including Alpine, We can use these in Charliecloud directly:

```
$ wget -O alpine.tar.xz 'https://github.com/alpinelinux/docker-al-
pine/blob/v3.16/x86_64/alpine-minirootfs-3.16.6-x86_64.tar.gz?raw=true'
$ tar tf alpine.tar.xz | head -10
./
./sys/
./srv/
./run/
./root/
./opt/
./mnt/
./media/
./media/usb/
./medis/floppy/
```

This tarball is what's called a "tarbomb", so we need to provide an enclosing directory to avoid making a mess.

```
$ mkdir alpine
$ cd alpine
$ tar xf ../alpine.tar.xz
$ ls
bin  etc   lib    mnt  proc  run   srv  tmp  var
dev  home  media  opt  root  sbin  sys  usr
$ cd ..
```

Now, run a shell in the container! (Note that base Alpine does not have Bash.)

```
$ ch-run ./alpine -- /bin/sh
$ pwd
/
$ ls
bin  etc  lib  mnt  proc  run  srv  tmp  var
dev  home  media  opt  root  sbin  sys  usr
$ cat /etc/alpine-release
3.16.6
$ exit
```

Warning: Generally, you should avoid directory-format images on shared filesystems such as NFS and Lustre, in favor of local storage such as `tmpfs` *and local hard disks. This will yield better performance for you and anyone else on the shared filesystem. In contrast, SquashFS images should work fine on shared filesystems.*

5 Key workflow operation: Build from Dockerfile

The other containery way to get an image is the *build* operation. This interprets a recipe, usually a *Dockerfile*, to create an image and place it into *builder storage*. We can then extract the image from builder storage to a directory and run it.

Charliecloud supports arbitrary image builders. In this tutorial, we use `ch-image`, which comes with Charliecloud, but you can also use others, e.g. Docker or Podman..

Note: `ch-image` *is a big deal because it is completely unprivileged which is important in environments like ours. Other builders run as root or require setuid root helper programs; this raises a number of security questions.*[2]

5.1 Exercise

We'll write a "Hello World" Python program and run it within a container we specify with a Dockerfile. Set up a directory to work in:

```
$ cd $CHORKSHOP
$ mkdir hello.src
$ cd hello.src
```

Type in the following program as "`hello.py`" using your least favorite editor.

```
#!/usr/bin/python3

print("Hello World!")
```

Next, create a file called "`Dockerfile`" and type in the following 4-line recipe:

```
FROM almalinux:8
RUN yum -y install python36
```

[2] See Charliecloud's Supercomputing 2021 unprivileged build paper: https://dl.acm.org/doi/10.1145/3458817.3476187

```
COPY ./hello.py /
RUN chmod 755 /hello.py
```

These four instructions say:

1. `FROM`: We are extending the `almalinux:8` *base image*.
2. `RUN`: Install the `python36` RPM package, which we need for our Hello World program.
3. `COPY`: Copy the file `hello.py` we just made to the root directory of the image. In the source argument, the path is relative to the *context directory*, which we'll see more of below.
4. `RUN`: Make that file executable.

Let's build the image:

```
$ ch-image build -t hello -f Dockerfile .
  1. FROM almalinux:8
[...]
  4. RUN chmod 755 /hello.py
grown in 4 instructions: hello
```

The `ch-image build` line says:

1. Build (`ch-image`) an image named (a.k.a. tagged) "hello" (`-t hello`).
2. Use the Dockerfile called "Dockerfile" (`-f Dockerfile`).
3. Use the current directory as the context directory .).

Now list the images ch-image knows about:

```
$ ch-image list
almalinux:8
hello
```

And run the image we just made:

```
$ cd ..
$ ch-convert hello hello.sqfs
$ ch-run hello.sqfs -- /hello.py
Hello World!
```

This time, we've run our application directly rather than starting an interactive shell.

5.2 Further reading

- Dockerfile reference: `https://docs.docker.com/engine/reference/builder/`

6 Key workflow operation: Push

The containery way to share your images is by *pushing* them to a container registry. (Above, we did the reverse of this operation: pulling from a registry.) In this section, we will set up a registry on `gitlab.com` and push the hello image to that registry, then pull it back to compare.

6.1 Set up registry

Create a private container registry:

1. Browse to `https://gitlab.com`.

2. Log in with your credentials. You should end up on your *Projects* page.
3. Click *New project* then *Create blank project*
4. Name your project “ws2023-07”. Leave *Visibility Level* at *Private*. Click *Create project*. You should end up at your project’s main page.
5. At left, choose *Settings* (the gear icon) → *General*, then Visibility, project features, permissions. Enable *Container registry*, then click *Save changes*. (Note this may already be enabled)
6. At left, choose *Deploy* → *Container registry*. You should see the message “*There are no container images stored for this project*”.

At this point, we have a container registry set up, and we need to teach ch-image how to log into it. However, GitLab has a thing called a *personal access token* (PAT) that can be used no matter how you log into the GitLab web app. To create one:

7. Click on your avatar at the top right. Choose *Edit Profile*.
8. At left, choose *Access Tokens* (the three-pin plug icon).
9. Type in the name “registry”. Tick the boxes *read_registry* and *write_registry*. Click *Create personal access token*.
10. Your PAT will be displayed at the top of the result page under *Your new personal access token*. Copy this string and store it somewhere safe & policy compliant. (Also, you can revoke it at the end of the workshop if you like.)

6.2 Push image

We can use “ch-image push” to push the image to gitlab.com

```
$ cd $CHORKSHOP
$ ch-image list
almalinux:8
hello
$ ch-image push --help
usage: ch-image push [-h] [--cache | --no-cache | --rebuild] [-a ARCH]
                     [--always-download] [--auth] [--debug] [--dependencies]
                     [--password-many] [-s DIR] [--tls-no-verify] [-v]
                     [--version] [--image DIR]
                     IMAGE_REF [DEST_REF]

copy image from local filesystem to remote repository
[...]
```

Note that the tagging step you would need for Docker is unnecessary here, because we can just specify a destination reference at push time.

When you are prompted for credentials, enter your e-mail address (that you use to log into gitlab.com) and copy-paste the PAT you created earlier. (Currently you will be prompted multiple times, which is a known bug.)

Note: <USER> is your GitLab moniker

```
$ ch-image push hello registry.gitlab.com/<USER>/ws2023-07/hello:latest
pushing image:   hello
```

```
destination:     registry.gitlab.com/$USER/ws2023-07/hello:latest
layer 1/1: gathering
layer 1/1: preparing
preparing metadata
starting upload
layer 1/1: bca515d: checking if already in repository

Username: <USER>
Password:
layer 1/1: bca515d: not present, uploading: 89.6/89.6 MiB(100%
config: f969909: checking if already in repository
config: f969909: not present, uploading
manifest: uploading
cleaning up
done
```

Go back to your container registry page. You should see your image listed now!

6.3 Pull and compare

Let's pull that image and see how it looks.

```
$ ch-image pull registry.gitlab.com/<USER>/ws2023-07/hello:latest hello.2
--auth
pulling image:   registry.gitlab.com/<USER>/ws2023-07/hello:latest
destination:     hello.2
[...]
$ ch-convert hello.2 hello.2.sqfs
$ ch-run hello.2.sqfs -- ./hello.py
Hello World!
```

7 MPI Hello World

We'll use a simple parallel application. The base image is a CentOS 8 image with OpenMPI already installed; OpenMPI takes about 30 minutes to build and install, so we don't want to take workshop time doing that.

7.1 Pull base image

This step is not strictly necessary, because "`ch-image build`" will pull the image if needed, but this particular image is quite large, so it may be useful to start the pull and then go have a break.

> *Note:* The reference for the base image is "`charliecloud/openmpi`" on Docker Hub, not plain "`openmpi`", because we uploaded it there.

```
$ ch-image pull charliecloud/openmpi openmpi
```

7.2 Build image

Create a new directory for this project, and within it the following simple C program called `mpihello.c`. (Note the program contains a bug; consider fixing it.)

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

int main (int argc, char **argv)
{
   int msg, rank, rank_ct;

   MPI_Init(&argc, &argv);
   MPI_Comm_size(MPI_COMM_WORLD, &rank_ct);
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);

   printf("hello from rank %d of %d\n", rank, rank_ct);

   if (rank == 0) {
      for (int i = 1; i < rank_ct; i++) {
         MPI_Send(&msg, 1, MPI_INT, i, 0, MPI_COMM_WORLD);
         printf("rank %d sent %d to rank %d\n", rank, msg, i);
      }
   } else {
      MPI_Recv(&msg, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
      printf("rank %d received %d from rank 0\n", rank, msg);
   }

   MPI_Finalize();
}
```

Add the following `Dockerfile`.

```
FROM openmpi
RUN mkdir /hello
WORKDIR /hello
COPY mpihello.c .
RUN mpicc -o mpihello mpihello.c
```

The instruction `WORKDIR` changes directories (the default working directory within a Dockerfile is `/`).

Build:

```
$ ls
Dockerfile  mpihello.c
$ ch-image build -t mpihello .
```

Note that the default Dockerfile is `./Dockerfile`; we can omit `-f`.

We need to convert the directory image to a squashball so we can run it on the compute nodes:

```
$ ch-convert mpihello mpihello.sqfs
```

7.3 Run the container interactively

We'll run this application interactively. One could also put similar steps in a Slurm batch script.

First, obtain a two-node allocation and load the Charliecloud module.

```
$ salloc -N2 -t 15:00
salloc: Granted job allocation 599518
[...]
```

Then run the application on both nodes in your allocation (Note the program contains a bug that causes warning messages; consider fixing it.):

```
$ srun --mpi=pmi2 ch-run mpihello.sqfs -- ./hello/mpihello
hello from rank 1 of 2
rank 1 received 0 from rank 0
hello from rank 0 of 2
rank 14 received 0 from rank 1
```

Win!

7.4 Run the container non-interactively with an sbatch script

Next, we'll run the same application non-interactively. We will put the same steps above, but in a sbatch script.

```
$ cat mpi_script
#!/bin/bash

#SBATCH --time=0:02:00          # walltime
#SBATCH --nodes=2               # number of nodes
#SBATCH --job-name=ch_mpi       # job name
#SBATCH --output=ch_out         # output file name

srun --mpi=pmi2 ch-run mpihello.sqfs -- ./hello/mpihello

$ sbatch mpi_script
Submitted batch job <JOB_ID>

$ cat ch_out
hello from rank 1 of 2
rank 1 received 0 from rank 0
hello from rank 0 of 2
rank 0 sent 0 to rank 1
```

Win !

8 Build Cache

ch-image subcommands that create images, such as build and pull, can use a build cache to speed repeated operations. That is, an image is created by starting from the empty image and executing a sequence of instructions, largely Dockerfile instructions but also some others like "pull" and "import". Some instructions are expensive to execute so it's often cheaper to retrieve their results from cache instead.

Let's set up this example by creating a new directory.

```
$ cd $CHORKSHOP
$ mkdir cache-test
$ cd cache-test
```

Suppose we have this Dockerfile a.df:

```
FROM almalinux:8
RUN sleep 2 && echo foo
RUN sleep 2 && echo bar
```

Let’s reset the build cache. Then on our first build, we get:

```
$ ch-image build-cache --reset
$ ch-image build -t a -f a.df .
  1. FROM almalinux:8
[ ... pull chatter omitted ... ]
  2. sleep 2 && RUN echo foo
copying image ...
foo
  3. sleep 2 && RUN echo bar
bar
grown in 3 instructions: a
```

Note the dot after each instruction’s line number. This means that the instruction was executed. You can see this in the output of the two `echo` commands.

But on our second build, we get:

```
$ ch-image build -t a -f a.df .
  1* FROM almalinux:8
  2* sleep 2 && RUN echo foo
  3* sleep 2 && RUN echo bar
copying image …
grown in 3 instructions: a
```

Here, instead of being executed, each instruction’s results were retrieved from cache. Cache hit for each instruction is indicted by an asterisk(“*”) after the line number. Even for such a small and short Dockerfile, this build is noticeably faster than the first.

We can also try a second, slightly different Dockerfile, `b.df`. Note that the first three instructions are the same, but the third is different.

```
FROM almalinux:8
RUN sleep 2 && echo foo
RUN sleep 2 && echo qux
```

Build it:

```
$ ch-image build -t b -f b.df .
   1* FROM almalinux:8
   2* sleep 2 && RUN echo foo
   3. sleep 2 && RUN echo qux
copying image
qux
grown in 3 instructions: b
```

Here, the first two instructions are hits from the first Dockerfile, but the third is a miss, so Charliecloud retrieves that state and continues building.

We can also inspect the cache:

```
$ ch-image build-cache --tree
*  (b) sleep 2 && RUN echo qux
| *  (a) sleep 2 && RUN echo bar
|/
*  RUN sleep 2 && echo foo
*  (almalinux+8) PULL almalinux:8
*  (HEAD -> root) ROOT


named images:     4
state IDs:        5
commits:          5
files:            6 K
disk used:        97 MiB
```

Here there are four named images: `a` and `b` that we built, the base image `almalinux:8`, and the empty base of everything `ROOT`. Also note how `a` and `b` diverge after the last common instruction `RUN sleep 2 && echo foo`.

For more examples: https://hpc.github.io/charliecloud/tutorial.html