

## **BEE Tutorial Exercises**

### **User Credentials**

Username: ecpuser

Password: ECP2019Houston

### **LOG INTO YOUR TUTORIAL SYSTEM**

Open a terminal on your laptop and SSH to the IP address provided. Use the Username and Password at the top of this page to log in. Please log into your system at least twice as multiple windows will be required.

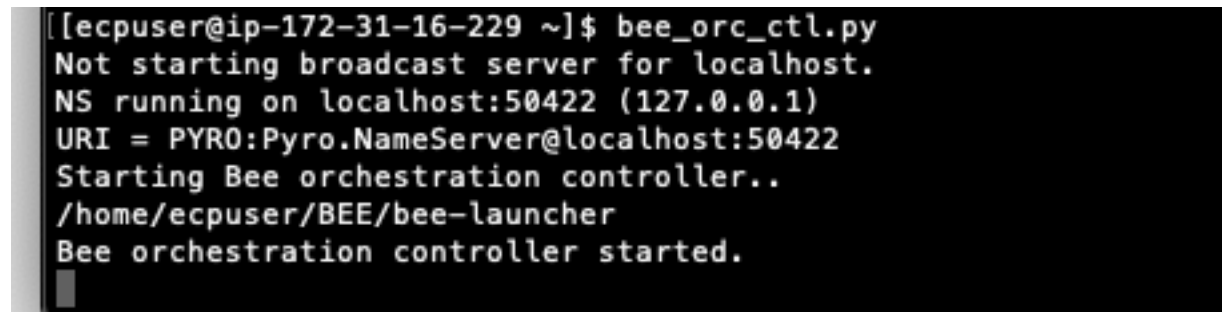
### **DOWNLOAD AND INSTALL BEE**

Downloading and installing BEE is a simple, straightforward process.

Type the following commands:

- 1) `git clone https://github.com/lanl/BEE.git`
- 2) `cd BEE`
- 3) `pip install --upgrade pip --user`
- 4) `./install.sh`
- 5) `export PATH=$PATH:~/BEE/bee-launcher`
- 6) `bee_orc_ctl.py`

If the installation is correct you should see something like:

A terminal window with a black background and white text. The prompt is [ecpuser@ip-172-31-16-229 ~]\$. The command bee\_orc\_ctl.py has been executed. The output is: Not starting broadcast server for localhost. NS running on localhost:50422 (127.0.0.1) URI = PYRO:Pyro.NameServer@localhost:50422 Starting Bee orchestration controller.. /home/ecpuser/BEE/bee-launcher Bee orchestration controller started. A cursor is visible at the end of the last line.

```
[ecpuser@ip-172-31-16-229 ~]$ bee_orc_ctl.py
Not starting broadcast server for localhost.
NS running on localhost:50422 (127.0.0.1)
URI = PYRO:Pyro.NameServer@localhost:50422
Starting Bee orchestration controller..
/home/ecpuser/BEE/bee-launcher
Bee orchestration controller started.
```

Press Ctrl-C to stop the BEE orchestration controller. Congratulations, BEE is successfully installed.

### **Exercise 0 – Charliecloud Familiarization**

First we will familiarize you with Charliecloud containers so you have a basic understanding of what BEE is doing behind the scenes on your behalf.

It is out of scope for this tutorial to teach you how to build container images so we have provided a number of pre-built images in `/usr/local/share/images`. Let's work through the steps necessary to use the Charliecloud container runtime to use these images. We'll start with the image you'll use in your first exercise.

*Charliecloud basics – prepare an image for use*

- 1) `ch-tar2dir /usr/local/share/images/openmpi-mpihello.tar.gz /var/tmp`

*Charliecloud basics – launch a container and inspect the environment*

- 1) `cat /etc/os-release | grep ^NAME`

```
[ecpuser@ip-172-31-16-229 ~]$ cat /etc/os-release | grep ^NAME
NAME="CentOS Linux"
[ecpuser@ip-172-31-16-229 ~]$
```

- 2) `ch-run /var/tmp/openmpi-mpihello -- /bin/bash`

- 3) `cat /etc/os-release | grep ^NAME`

```
[ecpuser@ip-172-31-16-229:/$ cat /etc/os-release | grep ^NAME
NAME="Debian GNU/Linux"
ecpuser@ip-172-31-16-229:/$
```

You can see that the command prompt and the Linux distribution changed once you ran the container. If you type “`uname -r`” in the container you should see 3.10.0-957.1.3.el7.x86\_64, which is the version of kernel running on your tutorial server. If you type “`exit`” to stop the container and type “`uname -r`” again you should again see 3.10.0-957.1.3.el7.x86\_64.

This is the major difference between Linux containers and virtual machines. In the case of Linux containers, you can have software from multiple Linux distributions running on top of the same kernel. In the case of virtual machines, the host system’s kernel is different than the virtual machine’s kernel. That’s because a virtual machine “boots” on a something called a hypervisor, which is a software representation of a physical computer. The hypervisor adds an additional layer of software between your application and the hardware. When it comes to HPC it is often that hardware that makes the system special. Linux containers, and in particular Charliecloud, enable a user to “bring their own software stack” to an HPC system and still reap the benefits of what makes HPC special.

If you are no longer inside the running Charliecloud container, redo step 2 above (`ch-run ...`) before continuing.

Now let’s run the MPI “Hello, World” example by hand.

- 4) `/hello/hello`

```
[ecpuser@ip-172-31-16-229:/$ /hello/hello
0: MPI version:
Open MPI v2.1.2, package: Open MPI root@a9f45c471d07 Distribution, ident: 2.1.2,
repo rev: v2.1.1-188-g6157ed8, Sep 20, 2017
0: init ok ip-172-31-16-229, 1 ranks, userns 4026532195
0: send/receive ok
0: finalize ok
```

This doesn’t look like many “Hello, World” examples that you’ve seen but it does give us some useful output. In particular the line starting “0: init ok” shows that we’re executing with one MPI rank.

Now let’s run the program again but this time launch it with the `mpirun` command.

- 5) `mpirun -np 2 /hello/hello`

```
[ecpuser@ip-172-31-16-229:/$ mpirun -np 2 /hello/hello
0: MPI version:
Open MPI v2.1.2, package: Open MPI root@a9f45c471d07 Distribution, ident: 2.1.2,
  repo rev: v2.1.1-188-g6157ed8, Sep 20, 2017
0: init ok ip-172-31-16-229, 2 ranks, users 4026532195
1: init ok ip-172-31-16-229, 2 ranks, users 4026532195
0: send/receive ok
0: finalize ok
```

This time we see two lines containing “init” each containing “2 ranks” which matches what we told the system to do by using “mpirun -np 2.”

Exit the container.

6) exit

Another bit of information is contained in the output from the examples above, the MPI version being used. You’ll see in each case that it says “Open MPI v2.1.2.” Let’s find out what version is installed *outside* of the container.

7) mpirun -version

```
[ecpuser@ip-172-31-16-229 ~]$ mpirun --version
mpirun (Open MPI) 2.1.5

Report bugs to http://www.open-mpi.org/community/help/
```

Here we see that the version of MPI inside of the container does not match the version outside of the container. The topic of MPI and Linux containers is too big to get into here, but we do want to point out that there are techniques to make it possible for a user to truly “bring their own software stack” and still run their applications like they are used to doing on bare metal HPC systems.

So far everything you’ve done with a container image and the Charliecloud runtime has been interactive. If you’re a user familiar with HPC batch systems you might be wondering, how would I do this in a job script? Let’s see an example of using Slurm’s srun command to execute the MPI “Hello, World” example, complete with multiple ranks.

8) srun --cpus-per-task=1 ch-run /var/tmp/openmpi-mpihello -- /hello/hello

```
[ecpuser@ip-172-31-16-229 ~]$ srun --cpus-per-task=1 ch-run /var/tmp/openmpi-mpi
hello -- /hello/hello
0: MPI version:
Open MPI v2.1.2, package: Open MPI root@a9f45c471d07 Distribution, ident: 2.1.2,
  repo rev: v2.1.1-188-g6157ed8, Sep 20, 2017
0: init ok ip-172-31-16-229, 2 ranks, users 4026532195
1: init ok ip-172-31-16-229, 2 ranks, users 4026532197
0: send/receive ok
0: finalize ok
```

This output looks a lot like the output in step 5) above, but in this case the Charliecloud runtime was invoked and the hello program was executed, complete with multi-rank MPI support, all in a Slurm command. This same command could be placed into a Slurm job script and submitted to a batch queue using salloc with the same results. Charliecloud’s ch-run command takes everything after “--” as the command to be executed

inside the container context. In step 2) the command you executed was “/bin/bash” which gave you a shell to do the interactive examples. In step 8) the command you executed was the MPI “hello” application.

In this example, Slurm has been compiled with support for launching MPI applications, which is a common way to build Slurm. Similar support exists in other resource managers such as PBS and Torque and is a very robust way to launch containerized MPI applications.

Let’s do a little clean-up before moving on to the next exercise.

```
9) rm -rf /var/tmp/openmpi-mpihello
```

### **Exercise 1 – bee\_orc\_ctl.py and bee\_launcher.py execution of MPI “Hello, World”**

In this exercise you will use the same container image, /usr/local/share/images/openmpi-mpihello.tar.gz that you used in Exercise 0, but this time you’ll use BEE to launch your application.

Change to the following directory in both of your terminal windows:

```
1) cd ~/exercises/1-mpihello
```

In here you’ll find two files, mpihello.beefile and mpihello\_run.sh. Open these files with your favorite editor (emacs, vi, nano, etc.) and inspect them. Each has a few placeholder comments where you will need to fill in the correct information. Make the changes and save the files.

In one of your terminals do the following:

```
2) bee_orc_ctl.py
```

You should see output like:

```
[ecpuser@ip-172-31-16-229 ~]$ bee_orc_ctl.py
Not starting broadcast server for localhost.
NS running on localhost:50422 (127.0.0.1)
URI = PYRO:Pyro.NameServer@localhost:50422
Starting Bee orchestration controller..
/home/ecpuser/BEE/bee-launcher
Bee orchestration controller started.
```

In another terminal do:

```
3) bee_launcher.py -l mpihello
```

```
[ecpuser@ip-172-31-16-229 mpihello]$ bee_launcher.py -l mpihello
Sending launching request.
[ecpuser@ip-172-31-16-229 mpihello]$
```

In the terminal where you started bee\_orc\_ctl.py you should see output like:

```

Bee orchestration controller: received task creating request
[mpihello] Verifying that Charliecloud is available...
0.9.7~pre+6243adc
[mpihello] Charliecloud launching
Preparing launch mpihello for nodes localhost
Launching local instance mpihello
[mpihello] Unpacking openmpi-mpihello to /var/tmp
[u'localhost'] Executing: ['ch-tar2dir', u'/usr/local/share/images/openmpi-mpihe
llo.tar.gz', u'/var/tmp']
[u'localhost'] creating new image /var/tmp/openmpi-mpihello
/var/tmp/openmpi-mpihello unpacked ok

localhost Executing: ['srun', '--nodelist=localhost', '--nodes=1-1', '--cpus-per
-task=1', 'sh', '/home/ecpuser/exercises/mpihello/mpihello_run.sh']
localhost 1: init ok ip-172-31-16-229, 2 ranks, userns 4026532197
0: MPI version:
Open MPI v2.1.2, package: Open MPI root@a9f45c471d07 Distribution, ident: 2.1.2,
repo rev: v2.1.1-188-g6157ed8, Sep 20, 2017
0: init ok ip-172-31-16-229, 2 ranks, userns 4026532195
0: send/receive ok
0: finalize ok

[mpihello] end event
[mpihello] Removing existing Charliecloud directory from localhost
[u'localhost'] Executing: ['rm', '-rf', u'/var/tmp/openmpi-mpihello']
[mpihello] Removed Charliecloud container openmpi-mpihello from /var/tmp
[mpihello] Has been successfully Terminated

```

Starting at the top, we see that the BEE orchestration controller received the task request from the BEE launcher. The mpihello.beefile specified the bee\_charliecloud backend launcher as “exec\_target.” We see the launcher verify that Charliecloud is indeed available. BEE is helpful in that it tells you what commands it is running. We see that the launcher is using the same ch-tar2dir command that you used in exercise 0 to prepare the openmpi-mpihello image in /var/tmp.

Once that is successful the output shows that srun is being invoked and the options being used. We know from exercise 0 that we expect /hello/hello to be run using ch-run, as specified in mpihello\_run.sh. Where your output may differ from that shows is in the number of MPI ranks. This exercise was prepared using a system with 2 CPUs, therefore the output above shows “2 ranks” whereas the systems used in the tutorial have 4 CPUs and should show 4 ranks in your output.

Following the output from /hello/hello we see that the bee\_charliecloud launcher does the node cleanup for us and removes the image from /var/tmp. If your workflow required running the same container image multiple times you can save runtime by specifying “remove\_after\_exec”: false in mpihello.beefile.

## **Exercise 2 – bee\_orc\_ctl.py and bee\_launcher.py execution of MPI PI**

In this exercise you’ll be writing your own beefile and run script to launch another containerized MPI application. This application uses the Monte Carlo method to calculate an approximate value for Pi.

Change to a new directory for this exercise:

```
1) cd ~/exercises/2-mpipi
```

This time your working directory is empty. You might want to copy and modify the beefile and run script from exercise 1, or at least open them for reference as you create new beefile and run scripts for this exercise.

Information you'll need:

**Container Image:** /usr/local/share/images/openmpi-mpipi.tar.gz

**Application in container:** /mpi\_pi/mpi\_pi

Write a new beefile and run script for the MPI Pi example and launch the task. Check the output from `bee_orc_ctl.py`. Does the output look like what you would expect? What approximate value of Pi was calculated? If you have any questions or would like someone to validate your results please ask one of the tutorial facilitators to have a look.

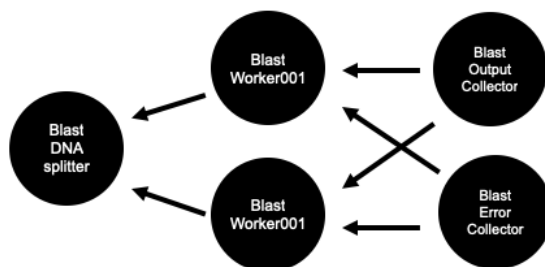
### Exercise 3 – bee\_orc\_ctl.py and bee\_composer.py execution of BLAST

In this exercise you will run a more complex workflow, learning about beeflow files and the `bee_composer.py` command.

Change to the working directory for exercise 3.

```
1) cd ~/exercises/3-blast
```

You might remember from the slides that BLAST has 3 distinct processing steps, the splitter, parallel workers, and parallel output collectors. The graph below represents the dependencies between these steps. Note that the 2<sup>nd</sup> and 3<sup>rd</sup> steps are done in parallel.



The beeflow, beefiles, and run scripts are all in place, but the dependencies still need to be defined in the beeflow file. Edit `blast.beeflow` with your favorite editor and properly define the dependencies between the various tasks. Then setup and run the workflow.

In one window:

```
2) bee_orc_ctl.py
```

In a second window:

```
3) bee_composer.py -f blast
```

This will immediately return to the command prompt.  
Watch the status of your workflow in the second window:

4) `bee_launcher.py -s`

You should see something like:

No.	Task Name	Status	Platform
1	blast-split	Finished	BEE-Charliecloud
2	blast-cleanup	Launching	BEE-Charliecloud
3	blast-output	Launching	BEE-Charliecloud
4	blast-worker001	Running	BEE-Charliecloud
5	blast-worker002	Running	BEE-Charliecloud
6	blast-output-err	Launching	BEE-Charliecloud

You can also track all of the stages in the workflow by examining the output in the window where you are running `bee_orc_ctl.py`. Your workflow is complete when all tasks are “Finished” or “Terminated.” Press `Ctrl-C` to stop the status monitor and return to the command line.

Take a look at the blast run scripts and you’ll see that there is a location where the input and output is directed. Once you find it in the scripts, `cd` to that location and take a look at the data it contains. Almost everything is ASCII and can be inspected. This is a demonstration of Charliecloud’s bind-mount capabilities. In the run scripts you’ll see the option “`-b $BLAST_OUT`” and references to “`/mnt/0/...`” in the `ch-run` command. This is how you get access to local filesystems such as your home directory and scratch directories inside of a Charliecloud container at run time. For more information consult the Charliecloud documentation.

You’ll also notice that there is a step not represented in the BLAST workflow graph, `blast-cleanup`. Do you have an idea why this step is necessary? We’ll give you the answer before we start the next section of the tutorial.

#### **Exercise 4 – bee\_orc\_ctl.py and bee\_composer.py execution of FleCSALE**

In exercise 4 you’ll put together everything you’ve learned today, beefiles, run scripts, and beeflow, to create your own example workflow. For this exercise you’ll be running the application FleCSALE. This example is very artificial and does not represent a useful workflow. It is only meant to demonstrate that you have learned how to put together the BEE components into a workflow that successfully executes as intended.

Change to the working directory for exercise 4.

1) `cd ~/exercises/4-flecsale`

Like in exercise 2, this directory is empty.

Here is information you’ll need for your beefiles and run scripts:

**Container Image:** /usr/local/share/images/laristra.flecsale.ubuntu\_charliecloud.tar.gz

**Arguments to ch-run:** "--no-home -b output --cd /mnt/0"

**FleCSALE in container:** /home/flecsi/flecsale/build/apps/hydro/2d/hydro\_2d

**Argument to flecsale:** "-m /home/flecsi/flecsale/specializations/data/meshes/2d/square/square\_32x32.g"

First write a beefile and runscrip to execute a single FleCSALE task using bee\_orc\_ctl.py and bee\_launcher.py. Once you are happy that it is running properly, write a beeflow that runs the same FleCSALE task twice serially with an off-line dependency. What would you need to change in your beefiles, run scripts, and beeflow to be able to run two FleCSALE tasks concurrently with an in-situ dependency? How can you clean up the image in /var/tmp when your workflow completes?

### **Bonus Exercise**

If you have completed all of the exercises and would still like to explore more, there are a couple other Charliecloud container images in your tutorial system that we haven't used today, beelanl.vpic.tar.gz and beelanl.cc-lammps.tar.gz. Feel free to use the Charliecloud commands you learned today (ch-tar2dir and ch-run) to run these images and applications.