Cheat Sheet: High-Performance Computing at the University of Leeds

R. Stone, S. Obute

April 10, 2019

Contents

1 Introduction

This cheat sheet is intended for new users of HPC at the University of Leeds. This is not intended to replace the useful links and training courses provided by HPC, which are included below:

- ARC Advanced Research Computing website
- ARC Training Courses
- **Any questions related to ARC usage should be sent to arc-help@lists.leeds.ac.uk**

This document is intended to get researchers started with HPC, and does not assume that readers have prior knowledge of command line, Linux or containers; however, neither does it provide a detailed explanation of them. It will show you how to run the commands and explain what they do.

You will need an ARC account in order to access ARC resources. You can apply for one here: apply for ARC account.

2 Logging in and basics

2.1 How to log in

Suppose you want to log onto ARC3. Open a terminal and type:

```
ssh sc16rsmy@arc3.leeds.ac.uk
```

Replace *sc*16*rsmy* with your own user name. You will be prompted for your University account password. If login is successful, you will see:

```
Last login: Fri Nov 23 19:38:06 2018 from vpn-host-40-024.leeds.ac.uk

Advanced Research Computing Node 3 (arc3)
------------------------------------------------------------------------

Information on using this facility may be obtained at the following URL:
http://www.arc.leeds.ac.uk

Please remember to acknowledge the use of ARC facilities in your
papers; details are on the website above.
------------------------------------------------------------------------

[sc16rsmy@login2.arc3 ~]$
```

There are two name nodes, or "master servers" that keep track of the whole cluster and keeps the file system. When you login, you will be on either *login*1 or *login*2. It doesn't really matter which.

2.2 Where is everything

By default, once you log in, you are in your home directory. You can view the directory you are in by using the Linux *pwd* or "print working directory" command:

```
[your_username@login2.arc3 ~]$ pwd
/home/home02/sc16rsmy
```

This is your **home directory**. You can create or move files and folders here. You will put all the scripts or code you want to run here. However, if you are using large amounts of data, you should NOT store them in your home directory. Home directories have a soft memory quota of 5GB, and a hard quota of 10GB. You can exceed the soft quota for about a week; if you exceed the hard quota you will be denied any resource requests.

You can easily view how much memory you have used using the Linux *quota* command:

```
[sc16rsmy@login2.arc3 ~]$ quota
Disk quotas for user sc16rsmy (uid 426110):
Filesystem blocks  quota  limit  grace  files  quota  limit  grace
nas-ufaservn1:/export/home/home02
3080372 10485760 11534336         9019      0      0
```

You should keep any large amounts of data in the */nobackup* directory. This is the shared parallel file system with 836TB of shared memory across all users. There is no quota but files unused after 90 days will be deleted (ARC will send you a warning email first), and if you use too much memory then you may get an email asking you to use less.

Navigate to the */nobackup* folder using the *cd* change directory Linux command:

```
[sc16rsmy@login2.arc3 ~]$ cd /nobackup
[sc16rsmy@login2.arc3 nobackup]$
```

If you haven't already, you should make yourself a directory here, and give it **the same name as your user name**. You will have read and write privileges inside the folder you create, so you can put all your data here. Make a directory using the *mkdir* Linux command:

```
[sc16rsmy@login2.arc3 nobackup]$ mkdir your_username
[sc16rsmy@login2.arc3 nobackup]$ cd your_username/
[sc16rsmy@login2.arc3 your_username]$ pwd
/nobackup/your_username
```

Put any big data, log files, checkpoints, etc. in here.

Note that data is not backed up, so keep an extra copy elsewhere in the unlikely event that anything happens to it. Other users can read other people's

files in the */nobackup* directory but cannot alter them.

2.3 How do I get files to ARC?

You have several options. If you are moving code or scripts which may frequently change, it is suggested to use a versioning control system such as Git or SVN. You will make your changes and tests on your local environment, and pull changes onto the remote environment on ARC.

If you are moving a lot of files to your */nobackup* directory, then you may want to use Secure Copy Protocol instead. You usually use SCP from your local machine. Suppose you are in the directory on your local machine which contains data you want to move to ARC. From the terminal, type:

```
$ scp * sc16rsmy@arc3.leeds.ac.uk:/nobackup/sc16rsmy/
sc16rsmy@arc3.leeds.ac.uk's password:
test1.txt                     100%  49    0.1KB/s  00:00
test2.txt                     100%  49    0.1KB/s  00:00
test3.txt                     100%  49    0.1KB/s  00:00
```

This copies all the files from the current directory onto ARC */nobackup/sc*16*rsmy/*.

2.4 I need software/package X. How do I get it?

First, check that it's not already available on the system. Check here: ARC Software for the list of applications, libraries, software, and programming languages you can load. If it's not there, I would suggest checking anyways to see if it's there but not on the website (it does happen):

```
$ module avail
```

If it's still not listed, then it's really not there. You have a few options depending on what exactly you need:

2.4.1 I need a Python package

You can either install it yourself using pip and the $--user$ flag:

```
$ pip install user name_of_package
```

Or with miniconda (but be aware that miniconda is separate from the the system versions of everything including the python interpreter, so you will need to install your own copies of everything through conda):

```
$ conda install name_of_package
```

2.4.2 I need a software or application X

Again, first check that it's not already there. If it isn't, then email arc-help@lists.leeds.ac.uk to see if it is already in an existing container, or what they suggest is the best way to install it.

3 Containers

If you have a unique set of packages/software/configurations that you need, it may be easiest to use a container. A container, simply speaking, is an isolated environment in which you can install anything you need without the risk of relying on system libraries that may change version, or may be the wrong version you need. You have full control over what is installed and available in a container. Example containers include Docker[1] and Singularity[2]. On ARC computers, you are allowed to run pre-built containers for your research. In this section, you are going to be introduced to the general steps you will take to get your container working on ARC.

You will need Docker installed on your local machine to be able to customize and build Docker containers. Using Singularity also requires that you install it on your local machine. However, if you are only interested in building your container using Singularity (or more commonly used approach of bootstrapping a docker image in your singularity recipe file), you will not need to install Docker on your local machine. You only need Docker if you want build Docker images. In the following sections, we give general guidelines for building your custom Docker container and converting it to a singularity image. After that, we then show how to bootstrap a Docker image and use it to build a Singularity container (we use this approach to build a Pytorch singularity container).

3.1 How to build one

If you are not familiar with containers at all, it is highly recommended to attend an ARC training course on containers.

Otherwise, if you already have some familiarity with containers, then the recommended work-flow is as follows:

3.1.1 Converting your Docker Image to Singularity

1. In your local environment (ideally, a VM or environment you can test in), create a custom Docker or Singularity file. ARC uses Singularity because it has better security; however, it is easy to convert a Docker image[3] file to a Singularity.

[1] `https://www.docker.com/`
[2] `https://www.sylabs.io/`
[3] There are hosts of Docker image files on Docker Hub, `https://hub.docker.com/`

An easy way to start is to find a Docker file on Docker Hub which contains the primary packages you need. You can then use it as a base to create your own.

2. Test your environment thoroughly to make sure it contains everything you need.

3. Convert your Docker file to a Singularity file using a conversion script, located on Github[4]

4. Move your Singularity image file to ARC using *scp* (see previous section 2.3). Container files are usually put in the directory /nobackup/containers, and are available to be used by all users (note that multiple instances of a single container can be used at once).

Any time you need to change your environment, you will need to rebuild your image file on your local environment, and copy the generated image file to ARC. Users do not have permissions to rebuild image files directly on ARC. If you anticipate making frequent changes to your container, you can consider making the container writeable, using the $-writable$ flag. This is NOT recommended for regular work flow.

3.1.2 Building Custom Singularity Containers

Sometimes, you may not find a singularity container[5] that meets your specification, and you will need to build one yourself. Also, you may want to modify a Docker image in Singularity before you use it on ARC. In this section, you will find some guidelines on building your own. For this example, we will build a Pytorch container.

The general steps:

1. Install Singularity on your local machine

2. Create a singularity recipe file[6]. The example below:

   (a) Bootstraps an Ubuntu 16.04 container from Docker hub

   (b) 'posts' all the command line arguments needed for installing Pytorch on an Ubuntu 16.04 machine. Note that we needed to install *Python3* and *pip3* first before installing *Pytorch* using pip3. The '%post' section usually contains all the terminal commands your software needs in order for it to be installed. You will need to install all dependencies for your software as well. Also note that when using 'apt-get', you should add the '-y' tag, so that your software will install without needing you to type 'yes' to approve its installation. On a final note,

[4] docker2singularity script`https://github.com/singularityware/docker2singularity`

[5] search for singularity containers at `https://singularity-hub.org/collections`

[6] For detailed information about how to create recipe files: `https://singularity.lbl.gov/docs-recipes`

sometimes, you may need to create folders within your container that 'binds' to actual file locations on the local machine it is being run on. This is not a requirement for Pytorch, but if you get complaints from your built container that a particular folder location is not found, you should create it within your '%post' section.

(c) Sometimes, your software may need you to manually set some environment variables. The appropriate place to do this is under your '%environment' section.

(d) in the '%runscript' section, you specify how you want your container to be run. Here, we just specified that your container should accept all the commands parsed to it from the terminal.

```
Bootstrap: docker
From: ubuntu:16.04

%post
    apt-get -y update
    apt-get -y upgrade
    apt-get -y install python3
    apt-get -y install python3-pip
    pip3 install https://download.pytorch.org/whl/cu90/torch-1.0.0-cp35-
        cp35m-linux_x86_64.whl
    pip3 install torchvision

%environment
    export LC_ALL=C.UTF-8

%runscript
    "$@"
```

3. Build the singularity image/container:

```
sudo singularity build <image-name> <recipe-file>
```

So if we name our Pytorch container 'pytorch100-cuda90.simg' and the recipe file we created in earlier as 'pytorch-pip', the command to build it is:

```
sudo singularity build pytorch100-cuda90.simg pytorch-pip
```

4. (optional) Create a test script to test if your pytorch container has access to GPU

```
import torch

if torch.cuda.is_available():
   print('cuda is available')
else:
   print('cuda is ~~NOT~~ available, using CPU')
```

5. Copy the Singularity image and optional test script to HPC (into your nobackup folder)

```
scp pytorch100-cuda90.simg cuda_test.py username.arc3.leeds.ac
    .uk:/nobackup/username
```

6. To test if your container works correctly on HPC, you should request an interactive shell on HPC, load singularity and cuda modules, then run/execute the container. In the example that follows, the 'singularity exec...' executes our singularity container. The '–nv' allows your container to have access to GPU if available; '-B' binds a folder on the host system (HPC in this case) to a folder location in the container, here we bind the /nobackup/username/ location on HPC to the /mnt location/folder/directory in our Pytorch container. The next term is the Singularity image/container we want to execute 'pytorch100-cuda90' after which we tell it which command to execute in it 'python3 /mnt/cuda_test.py'. Here, the path to the python script is accessed through the directory we've bound to /nobackup/username (location of the script on our HPC)

```
qrsh -l coproc_k80=1,h_rt=0:10:0 -pty y /bin/bash -i

module load singularity
module load cuda

singularity exec --nv -B /nobackup/username/:/mnt pytorch100-
    cuda90.simg python3 /mnt/cuda_test.py
```

If you get 'cuda is available' response on your shell from cuda_test.py, then you now have pytorch container with access to GPU on HPC.

4 The scheduler

4.1 Request an interactive session

Now that you are familiar with the name nodes and ways of setting up an environment, let's consider resources: CPU's, or GPU's. The resources available differ depending on the system, which you can see on the ARC website.

Using ARC3 as an example, there are k80's and p100's available. There are 2 k80's per machine, or 4 p100's. You can request an interactive session on one

of these resources as follows:

```
[sc16rsmy@login1.arc3] ~$ qrsh -l coproc_k80=1,h_rt=3:0:0 -pty y /bin/
    bash -i
```

- **coprok_k80** indicates a resource request for half GPU, which on ARC3 is 12 cores, 64 GB and one k80. If you request two, you will be using the entire GPU. It is good practice to request only what you need.
- **h_rt** is hard runtime, requesting usage for a specific length of time (hh:mm:ss). The maximum time allowed is 48 hours (48:00:00). Your session will exit automatically after the time runs out.
- **-pty** makes the scheduler allocate the GPU to you.
- **/bin/bash** links the session to the right bash directory.

If the request was successful, you will see your prompt change to the machine you are now on:

```
[sc16rsmy@db12gpu1.arc3] ~$
```

If you are on a GPU resource, make sure to load the *cuda* module in order to use the GPU:

```
[sc16rsmy@db12gpu1.arc3] ~$ module load cuda
```

4.2 Submit a job

For most jobs, you won't want an interactive session; you'd rather queue up the job and be informed when it has finished. First, put everything that is needed to set up your environment and run your code in a Bash script. Suppose I want to run a Python file which trains a model. I create a new script, containing:

```
#!/bin/bash

# Prepare environment
module load cuda
module load python
module load python-libs/3.1.0

# Python script with all parameters
python ../code/train_model.py 0 1e-4 2000 15
```

Now you can queue up this job to be run. The parameters for *qsub* are the same as for *qrsh*:

```
$ qsub -cwd -V -l coproc_p100=1 -l h_rt=2:00:00 -m be train_model.sh
```

There are a few new parameters:

- **-cwd** executes the script from the current working directory
- **-V** verbose output
- **-l** allows specifying the resource type (i.e., coproc_p100=1)
- **m** look for mail options
- **b** send the user (by default, the user name printed when typing the Linux *whoami* command, leeds.ac.uk) at the *beginning* of the job, when it is started
- **e** send the user an email when the job is completed. You may replace "be" with "n" if you do not want to receive any emails.

Optionally, you may put the *qsub* parameters **inside** the Bash script itself. The following results in the same outcome:

```
#!/bin/bash
#$ -cwd
#$ -V
#$ -l coproc_p100=1
#$ -l h_rt=2:00:00
#$ -m be

# Prepare environment
module load cuda
module load python
module load python-libs/3.1.0

# Python script with all parameters
python ../code/train_model.py 0 1e-4 2000 15
```

And then run the script simply with:

```
$ qsub train_model.sh
```

4.3 SGE task arrays

Often, you may want to run multiple instances of a script with different sets of parameters. SGE task arrays are helpful for these types of jobs, as they allow you to queue up a single job with a single **qsub** command, and also delete all jobs with a single **qdel**.

An example usage from the ARC3 website:

```
# Use current working directory and current modules
#$ -cwd -V
```

```
# Request Wallclock time of 6 hours
#$ -l h_rt=06:00:00

# Tell SGE that this is an array job, with "tasks" numbered from 1 to 150
#$ -t 1-150

# Run the application passing in the input and output filenames
./myprog < data.$SGE_TASK_ID > results.$SGE_TASK_ID
```

Please see ARC3's complete documentation on task arrays: `https://arc.leeds.ac.uk/using-the-systems/why-have-a-scheduler/advanced-sge-task-arrays/`

4.4 View resource status

You can tell the status of all submitted jobs using the *qstat* command:

```
[sc16rsmy@db12gpu1.arc3] ~$ qstat
Job ID     Username Queue        Jobname    Limit State
-----------------------------------------------------------
240        sc16rsmy   db12           foo.sh       27:00  q
379        sc16rsmy   db08          boo.sh       27:00  r
```

A job state of "w" means that the job is getting queued up; "q" means that the job is waiting in the queue. A state of "r" means that it is running. You can manually terminate a job:

```
[sc16rsmy@db12gpu1.arc3] ~$ qdel job_id
```

5 Using a Jupyter notebook

If you want to test or play around with code on a GPU, with a graphical user interface, you can run it inside (or outside of) a container. This example shows how to set up a notebook inside a resource using the deeplearn_1.img container.

First, we will request an interactive session with half a GPU:

```
[sc16rsmy@login2.arc3] ~$ qrsh -l coproc_k80=1,h_rt=1:0:0 -pty y /bin/
    bash i
[sc16rsmy@db12gpu1.arc3] ~$
```

In this example, I was allocated db12gpu1. Next, load the modules you need. In this case, I will load singularity because I will be running the notebook inside a container:

```
module load cuda
module load singularity
```

Finally, start the Jupyter notebook inside a singularity shell, without a browser:

```
singularity shell --nv /nobackup/containers/deeplearn_1.img -c "export
    XDG_RUNTIME_DIR="";/opt/conda/bin/jupyter notebook --notebook-dir=
    $HOME --ip='*' --port=8887 --no-browser"
```

Note the port name which you have chosen; in this case, 8887. The port needs to be unique. If you get a permissions error, you may have forgotten the –nv flag and the export command.

Now, in a separate terminal window from your local machine (NOT logged on to ARC), forward the port to your local machine port:

```
$ ssh -L 11111:db12gpu1:8887 sc16rsmy@arc3.leeds.ac.uk
```

Replace db12gpu1 with the GPU you were assigned, and change sc16rsmy to your user name. It will ask for your password and if successful, will log you into ARC.

Now, navigate to your local browser at the port you specified. You should see the Jupyter notebook appear as shown below:

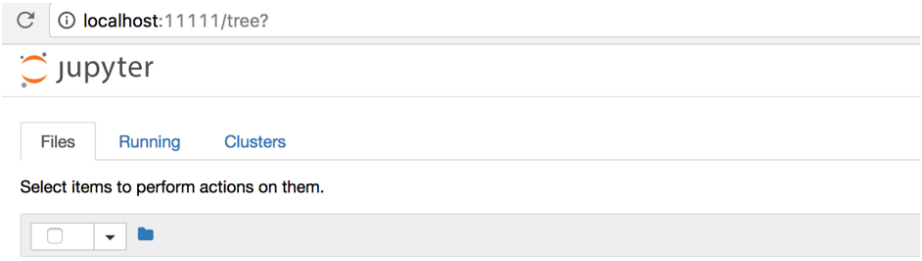

Figure 1: The notebook running inside the container on ARC3, accessed from your local browser.

Note that it is also possible to run a Jupyter notebook outside of a container (load the modules you need before starting the notebook), and on the name node. For example, if you do not need a GPU and simply want to edit some code using a GUI, you may run the notebook directly on a name node.

6 CPU-only I/O tasks

Sometimes you may want to do tasks which are primarily I/O (i.e., reading and writing files, but not doing much calculation on anything.) These kind of tasks

require no GPU, but may take awhile. You may run these on the name nodes directly, but be careful not to hog the CPU since other users may be using them. The following two sections are tips about how to do such tasks in a nice way (for yourself and for others).

6.1 screen

Suppose you want to do some long I/O task specified in a Bash script, such as copying lots of data back and forth from somewhere. You may not want to stay logged in the entire time, and don't want to risk losing network or connection and having to restart the script. You can use the Linux screen command:

```
$ screen
```

It will clear your window and environment. Now, run the script:

```
$ ./long_task_x.sh
```

To detach from the screen, type Ctrl+A d. You will see:

```
[detached from 84519.pts-46.login1]
```

You can view all the active screens using the $screen - ls$ command:

```
$ screen -ls
There is a screen on:
84519.pts-46.login1 (Detached)
1 Socket in /var/run/screen/S-sc16rsmy.
```

To re-attach to the first screen, type:

```
$ screen -r screen_ID
```

If you do not provide a screen ID, it will automatically re-attach you to the first screen. Once you have detached from a screen, you can log out from ARC, turn off your computer, and do anything you like. You can log back in and re-attach to it later.

6.2 Being nice

To not hog CPU on a name node, you may want to use *nice*. Nice is a Linux command which assigns priorities to jobs; i.e, if you are running a long I/O task and no one else is using the server, you can have all the CPU; otherwise, if it's being used a lot, your task will be assigned a lower CPU priority so as not to hog all the resource.

You can read about what priority to use here: about Nice

```
nice -n 10 python my_long_script.py
```

Like any other command, nice can be used inside of a screen.

7 Conclusion

This document was written by Rebecca Stone and Simon Obute, with help from Martin Callaghan, Mark Dixon, and Leo Pauly.

Please contact ARC staff directly for any specific issues related to ARC usage.