CC Primer: Compute Canada for geoscientists in a rush

Ricardo Barros Lourenco

Last revision: 2023-06-13

Contents

Chapter 1

Front Matter

1.1 Copyright

Otherwise stated in the text, this content is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License.

1.1.1 How to cite this work

Please use this BibTeX fragment:

```
@book{BarrosLourenco2022,
  title     = "CC Primer: Compute Canada for geoscientists in a rush",
  author    = "Barros Lourenco, Ricardo",
  year      = 2022,
  doi       = "10.5281/zenodo.5937906"
  note      = {\url{https://ricardobarroslourenco.github.io/CCprimer/}(visited yyyy-mm-dd)}
}
```

Obs.: Note that the DOI is of a previous version (10.5281/zenodo.5937906) when comapred to the badge displayed on this website, because it relates to all versions of this document, rather than a especific one.

1.2 Cronology

2023-06-07 - v0.3 - Adds Jupyter description, and BASH screen tool usage

2023-05-23 - v0.2 - Adds description on how to run Rstudio Server on Alliance Canada Premises.

2022-01-31 - Beginning of v0.1. (This version is quite unstable, and versioning during this stage will be limited).

Chapter 2

Version Control Systems

Version Control Systems Wikipedia contributors [2002] (VCS - also named as *Revision Control*, *Source Code Control*, *Source Code Management*) are Computer Systems traditionally used to manage the complexities of software development, majorly Computer Systems involving multiple software modules, and large development teams.

Such systems help to organize such development, especially when collaboration of multiple developers is done, and several times people are doing code changes in a same, or close, part of the code, which may break it, or even generate unexpected results that would not be traceable.

2.1 VCS: Local repository vs. Remote repository

Common VCS's (ex.: CVS, SVN and Git) always are client-server systems.

The VCS client is not merely a means to access a central repository hosted at the VCS server, but actually is part of running checks and balances that are necessary when adding code to the central repository. A situation that demands that is when multiple users are doing a change in a same piece of code, and checking it only at the central repository may complicate more a situation already complicated (simultaneous contribution). Therefore, it hosts, locally a local copy of the codebase, which reflects a view of that codebase in time (more preciselly when that local user retrieved a copy from the central repository for editing).

Once the changes are done, the user saves a local copy of these at the local repository, and if these meet the criteria for changes (and this may vary a lot among systems), the user is able to persist this change at the local repository,

as a new version of the codebase. This process is known as *commit* (in this case, a local one).

2.2 Git

Git is a contemporary VCS that is used as the backbone of operations run on GitHub. So, when planning to operate with GitHub, it is almost certain that you will need to install a Git client on your machine.

We can say *almost* because GitHub provides an own client, GitHub Desktop which provides a Graphical User Interface(GUI) to a git client.

We will not cover such GUI usage since on Compute Canada you will not have access to GitHub Desktop on their machines, but only to a regular Git client (that you can use to access either GitHub, or another Git compliant server such as a private repository built with GitLab).

2.2.1 Installing a git client on your machine

2.2.1.1 Ubuntu (or any other Debian-based environment)

First update your Operating System (OS) repository indexes:

```
$ sudo apt-get update
```

Then proceed to install:

```
$ sudo apt-get install git
```

Note: If you use Ubuntu, git is already provided as a base repository on this distribution. If you use another linux, and this does not work for you, please open an issue on this project and use the label *requests*, and I will try to solve and include here for future reference.

2.2.1.2 Apple

Apple is an OS that not use repositories by default, and git is not provided in the main set of software. So you would have three main options to install git:

- Install Xcode (*Preferred*): Apple has a software development kit (SDK) named XCode which provides git among several other software for MacOS and iOS software development. The advantage of using XCode´s git distribution is that it comes adjusted for your OS and is supported by Apple, whcih avoids conflicts and security issues.
  - To install XCode (and git by consequence), open the App Store, and search XCode as a software developed by Apple, and install it. To test, open the Terminal App, and run *git*. You should receive an output of basic description of commands available on git.

- Install Homebrew, and then install Git (if you are a Homebrew user, or uses a lot of *.nix software not supported by Apple it is useful):
  - Download and install homebrew from their website;
  - Install git as:

```
$ brew install git
```

  - In the terminal, run *git* and you should receive an output of basic description of commands available on git.
- Install a standalone version of git (*not recommended*): You can download and install from git´s website a standalone client (I will not be covering this approach, because git will not be updated via this kind a install, posing a security risk and install this at your own risk - and effort!)

2.2.1.3 Windows

Since Windows works with standalone installations, the install follows as this:

- Go to the Git website and download the latest client (download the standalone version, unless you have severe storage limitations on your machine);
- Once downloaded, run the installer with admin permissions and follow the default installation;
- When installed, open PowerShell (PS) (or the command prompt, if you do not have PS installed), and run *git* and hit enter. You should receive an output of basic description of commands available on git.

2.2.2 Setting up your local git client

Once you have your git client up and running, you need to setup the access credentials to connect to a git repository, such as GitHub or GitLab for example.

To do so, run on your bash terminal (or command-line if on windows):

```
$ git config -- global user.name "Your full name"
```

Note: On quotes you need to specify a name (depending on the environment, it should be your full name, or an alias, such as employee number, for example). This command has a global scope, so all users on your machine would have the same setup.

Then you need to specify an e-mail address (if on GitHub, it is preferred that is linked to your account):

```
$ git config -- global user.email "Your e-mail address"
```

Then check, if all values were set:

```
$ git config --list
```

You should see an output that reflects those previously set name and e-mail values.

2.3 GitHub

GitHub is a Web collaborative Version Control service based on a Git platform. It was created in 2008, and as of today is the largest repository of source code in the World. Aside of VCS capabilities, it provides other computational services, such as Continuous Integration (CI) and Continuous Deployment (CD), Web Page Hosting on GitHub Pages, security solutions, a software marketplace, among many others. It was recently acquired by Microsoft, as one of the largest transactions in the tech domain.

GitHub has several different tiers of access, from basic free accounts up to corporate ones.

A main advantage for academic users would be using the GitHub Education which includes a free GitHub PRO account and several software for free (or services credit hours) while you are enrolled in an academic institution. GitHub Education also have different tiers of users aiming students, teachers, and the academic institutions itselves. Once you create a simple free GitHub account, you can return back to GitHub Education, and request to be enrolled in the program.

2.3.1 Creating an account

To create an account is simple:

- Go to GitHub pricing page, select the Free tier, and click *Join for Free*;
- Just follow the prompts to create your personal account.
- You should receive an e-mail from them on the registered e-mail account, to verify your identity. If you fail to do so, your account would be basically useless.

Note: Since I have created my account some years ago, I am just using GitHub's user documentation as reference. If things get complicated in this step, please let me know, and I will expand here.

2.3.2 Insert GitHub credentials on your local Git

Once you have your GitHub account created, and verified, it is time to setup these credentials on your local Git client. While logged on your GitHub account you should:

- Click on your *name/avatar/photo at the upper right corner of the screen*, and then click on *Settings*;
- Then click on *Developer Settings* (it is the last item on the bottom of items at the left panel);

- Now click on the bottom item, *Personal access tokens*;
- Then, click on *Generate new token*, you will be requested again for your password on this step;
- Now you will be required to fill in some info:
  - *Note*: This will be a name of this Token. I often create a Token per software+device I use, then someone can write "Rstudio on Laptop", to differentiate from "Rstudio on laboratory desktop".
    - Isolating tokens on different (software,device) pairs helps isolating accesses to your GitHub account, and is important to contain a security breach. If someone is able to fetch one of your tokens, you will know from which machine and which software on it came from.
  - *Expiration*: You should define a expiration date for your token. Choosing a date is an open ended question, but ideally systems exposed to the Web, or multiple users such an HPC cluster, should be changed frequently.
    - *Avoid at all means to use the No expiration* mode, because you never want to forget unattended keys of your GitHub (ex. You graduate, and your key is left on a machine that another student will use in the lab).
  - *Select scopes*: Perhaps this is the trickiest setup. The definition of a Token is actually a definition of a OAuth Token. On GitHub, this implies on being able to select all options that a user has in terms of operations on the platform, and actually, narrowing down to what a user wants to grant permission for in such Token.
    - Note: It is tempting to grant *all permissions* to a single Token, but the user should ask it that is really necessary. In one end, granting all permissions is too permissive, and being as problematic as having a token with no expiration data.
    - To look into what each scope covers, please look into this page.
- Once you have generated your token, treat is as a password (even in terms of security/sharing/etc.). You should not persist a copy of it, but only use for your local git setup. Even if you loose access to it, you can always revoke that token, and create a new one.
- To set your local git to access GitHub (remind that you need first to set global variables), you just need to access GitHub with it. A simple way, that will be further explained in more detail would be making a local copy of a repository using *git clone*:

```
$ git clone https://github.com/a_very_weird_user_name/a_more_weird_user_project_name.git
```

A concrete example would be the source-code of this document:

```
$ git clone https://github.com/ricardobarroslourenco/CCprimer.git
```

- Once done, your local git should request:
  - User-name: The one you have set for your GitHub account;

  - Password: Your token.
- After this, it should make a local copy of that cloned repository (more especifically this copy will be stored at the directory path you have run the clone command - in bash / terminal / command line).
  - Note: If you have not been requested a username/password, maybe you have already a GitHub account already set on your machine (so please note if any errors occur when trying to access GitHub with git, no username + password is required - if these occur, they will be output on your bash / terminal / command line)
- Finally, to persist the token on your local install, run:

```
$ git config --global credential.helper cache
```

- If you need to clean-up the token(s) installed:

```
$ git config --global --unset credential.helper
```

2.4 A combined usage of Git-GitHub

This section describes a usage of Git-GitHub intended for scientific usage. Therefore, is important to remark that there is still no consolidated practice on such application, considering that these tools were not meant for academic usage, but rather for a software engineering one. Such application varies across research groups and scientific domains, and on a best effort, we will summarize some good practices and update as needed.

2.4.1 Basic commands to update a repository

After cloning a public repository, a user may want to update it in several ways. A very common HPC scenario is when the user is testing code in production, in a cluster environment, while debugging. Other scenarios are related to persist small data outputs - considering the repository storage limitations.

Given that, users should do, once saved their changes to the local git is to *add*, *commit* and then *push* changes.

2.4.1.1 Git Add

When the user create or delete file(s) it is necessary to inform `git` that such addition/deletion happened via:

```
$ git add folder_url
```

If you want to update the entire project, you need to go via bash to the root of the git project, and then:

```
$ git add .
```

2.4.1.2 Git commit

After eventual addition/deletion(s) are informed to git, then, it is necessary to inform updates and the nature of these. This can be done via:

```
$ git commit -m "message describing the commit"
```

Ideally the message would describe what was done. Important to note that this could be done multiple times, because this is only persisting local changes (on the machine which the client is installed), not global changes. It is good practice to group "same topic" commits, as it is easier to do a rollback of such changes, avoiding to impact changes not related to a found issue.

2.4.1.3 Git push

Once the local repository changes are finalized, the user needs to synchronize these local changes with the main git(like)/GitHub repository. Only after this, these changes would become available to all users.

Often the repository on github has a `main` branch, therefore, the command is:

```
$ git push -u origin main
```

In which `origin` is the local repository, and `main` is the branch you want to update in the git server. These may be different, depending on your project characteristics.

Chapter 3

Compute Canada

Note: This section is under heavy work. For now, please refer to Compute Canada's Wiki.

3.0.1 Setting up SSH on Compute Canada portal

From January 2022 forward, Compute Canada is enforcing that is mandatory to use a key to use SSH with their premises (ex.: accessing login nodes). For this, you need to create a public-private SSH key, and *drop* the public key at the Compute Canada user portal, under your account settings.

3.0.1.1 Creating a SSH key

Some description of this is found at the Compute Canada documentation Compute Canada [2002] . Is good practice to read it, since it covers some specifics.

To create a SSH key you need to use *ssh-keygen*. On windows, you can access it through PowerShell, on MacOS using Terminal, and on any Linux distribution via a shell terminal (such as Bash, for example).

In a general way, you can create a new key pair with this command:

```
$ ssh-keygen -C 'compute canada systems' -f computecanada-key -t rsa -b 4096
```

The parameters refer to:

- *-C*: A label for the key being generated, useful when you have multiple SSH keys in a machine;
- *-f*: The name of the key file. Once run, ssh-keygen will output a file with such name without extension which is the *private-key*, and another file, with a same name, but with extension _*.pub_, corresponding to the *public-key*.

- *-t*: Represents the choice of encrypting scheme, which on this case is a RSA one, with *4096* bits as the key size.

Once you hit the command, you should have on the folder you are running these two files: - *computecanada-key* (without file extension): corresponds to the private key; - *computecanada-key.pub*: corresponds to the public key.

WARNING: The private key is equal to a regular password to the system you are using it to access. Therefore, it is recommended that you assign a key password (it will be requested when you are generating the key), to assure that if such key is lost/copied/etc, nobody would be able to access this system impersonating you. A series of good practices for ssh keys may be found at this post.

3.0.2 Depositing the public key on Compute Canada

Once created the key pair, you need to deposit the *public key* on your Compute Canada account. A main tutorial is provided here but you can proceed as follows:

- Login at your Compute Canada account, you should see a screen as such:

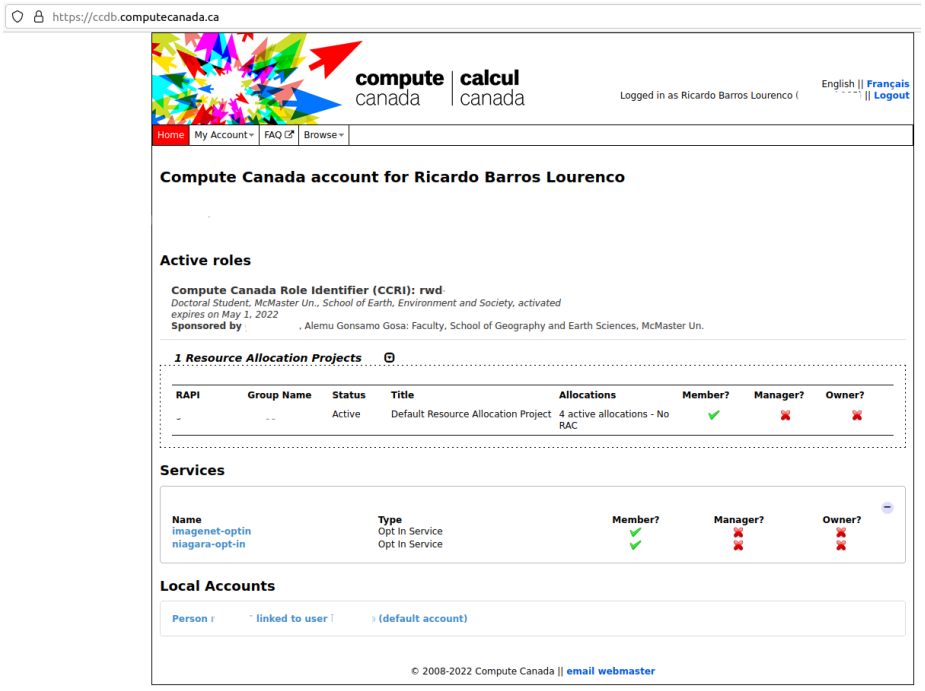

Figure 3.1: Landing page of your Compute Canada account

- Enter in the *My Account* tab, then click *Manage SSH Keys*:

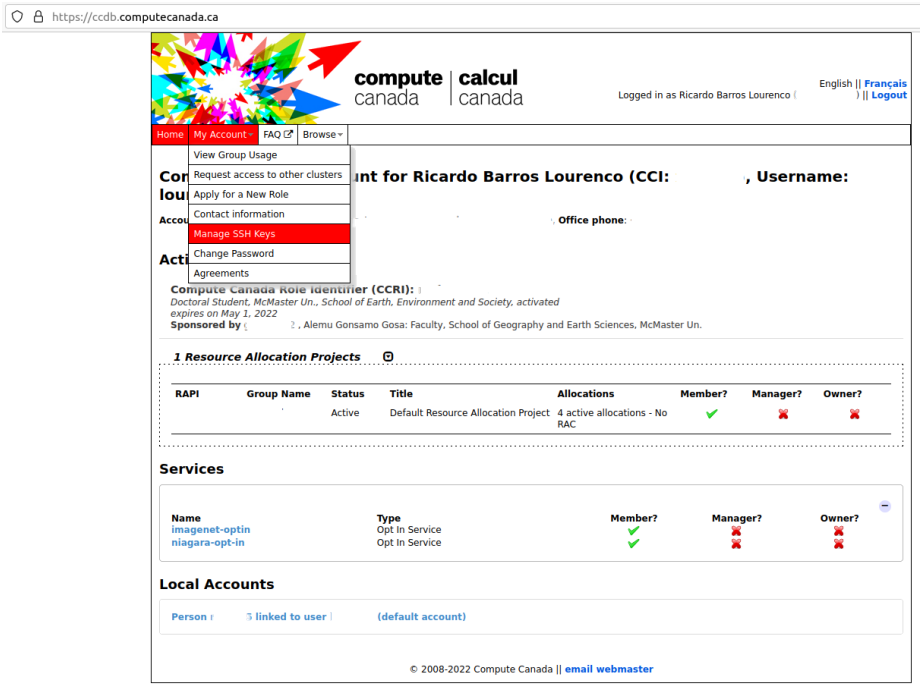

Figure 3.2: Manage SSH Keys menu

- Then you should have the SSH Key management options:

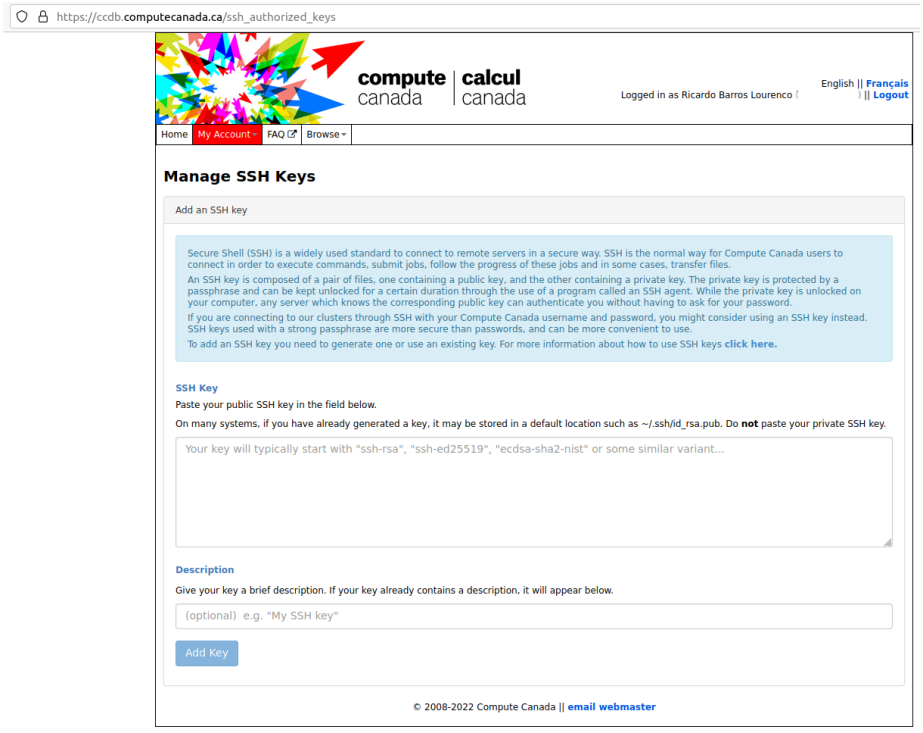

Figure 3.3: SSH Key Management options

- Now insert the content of your SSH *public key* in the first field. To do so, you need to open your *.pub* file with a simple text editor (ex.: Notepad on Windows, TextEdit on MacOS, Gedit/Vi/Emacs on Linux - Do not open with text processing software, as MS Word!) select all content and copy-paste into the first text box.

- Then you should assign a *Description* to such key. I recommend that you use different keys for every system you use to connect to Compute Canada Systems, and this field should reflect each of these systems. For example, a description as such "*Desktop Machine at the Laboratory*", or "*Personal Laptop*" would be enough to identify where these keys lie (especially important if you get hacked and someone steals your key).

- Then hit *Add Key*. This operation will put your public key on every machine of Compute Canada, and may take up to 30 minutes to be online.

3.0.3 Accessing a Compute Canada *login node*

Once your public key is loaded and synchronized in all machines, you may login a compute node using a terminal (On Windows, you need to use PowerShell; on MacOS, Terminal; and on Linux, your preferred Shell, such as Bash):

```
$ ssh -i ./computecanada-key your-compute-canada-username@MMMMMM.computecanada.ca
```

The command opens a SSH terminal session with the following parameters:

- *-i* : The path and the name of your *Private Key*;
- *your-compute-canada-username* : The username you have set for your Compute Canada account, when registering with them for an account;
- *MMMMMM* : One of the Compute Canada clusters (for example, Cedar, Graham, or Beluga)

Note: We will provide a more extensive list of computing environments later, but for now you may want to refer to this presentation.

3.0.4 Running batch jobs

If you want to run a R-language batch job, please take a look on the CCrecipes repository.

More specifically at the `slurm/R/r_batch_standalone.sh` file, we have:

```
#!/bin/bash

### Sets shell for parallel program (no distribution - no MPI)
### Inspired by the current documentation available on CC Wiki.

#SBATCH --account=def-someacct             # replace this with your PI account
#SBATCH --nodes=1                          # number of node MUST be 1
#SBATCH --cpus-per-task=4                  # number of processes
#SBATCH --mem-per-cpu=2048M                # memory; default unit is megabytes
#SBATCH --time=0-00:15                     # time (DD-HH:MM)
#SBATCH --mail-user=yourname@someplace.com # Send email updates to you or someone else
#SBATCH --mail-type=ALL                    # send an email in all cases (job started, job ended,

### Load library modules
module load gcc/9.3.0 r/4.0.2

### Load r-packages (builds locally) - comment, if unecessary to install new libraries
#R install_script.R

### Export locally installed packages
# TODO: check if this path holds
export R_LIBS=~/local/R_libs/
```

```
### Run main_job.R - or any batch script necessary
R CMD BATCH --no-save --no-restore ~/main_job.R
```

Some requirements:

- The R file `main_job.R` needs to be at the root of your user folder.
- Also the script needs to be executable (command runned at the same folder as `r_batch_standalone.sh`:

```
$ chmod +x r_batch_standalone.sh
```

- Finally, you can submit this batch job as:

```
$ sbatch r_batch_standalone.sh
```

Once submitted, you should receive a job number at the console. Once its status change at the scheduler (job started, job ended, job aborted), you should receive an e-mail.

If getting into running issues, take a look on the errors by inspecting the output files which will be saved at the same folder you submitted the job and have the name `slurm-xxxxxx.out` in which `xxxxxx` is the job number.

Chapter 4

BASH

On this section we will write some hints on the Bourne Again Shell (BASH), a command interpreter used in several *nix systems, such as Linux, Unix, MacOS and recently incorporated into Windows through PowerShell. This will be more of a collection of hints, and not substituting a full knowledge of BASH and the Operating System in which is running.

4.1 Screen

Screen is a tool you can use to keep a ssh session run through BASH alive. Why it is useful(?):

1. When you have unreliable internet connections connections (ex.: Wifi, cellphone);
2. Power outages;
3. Your local machine freezes, or have any other issue, and your connection is killed;
4. Any random reason that blocks your local terminal to access the remote ssh session.

4.2 How to use it

Some useful description is available in this website, and on the app documentation. But you can proceed as follows:

4.2.1 Creating a session

You create a new session, named *session_01* as:

```
screen -S session_01
```

4.2.2 See all available sessions

```
screen -ls
```

You probably will see something like this:

```
There are screens on:
        6617.session_00      (09/26/2019 04:35:30 PM)    (Detached)
        1946.session_01         (09/26/2019 02:51:50 PM)    (Detached)
2 Sockets in /run/screen/S-shs
```

4.2.3 Reattach to a previously established session

Let's say you want to attach to *session_00*:

```
screen -r session_00
```

4.2.4 Finishing a session

Let's say you want to finish *session_01*. You first enter in another session, and:

```
screen -XS 1946 quit
```

If the session you are aiming to finish is the last one, then:

- **ctrl + A**;
- **k**;
- and **y**.

Chapter 5

RStudio in Alliance Canada

Note 1: This refers to an experimental usage of RStudio for HPC. This approach has no official support from the Alliance, and the responsability is solely to the user. An official R environment is provided, with support, by the Alliance at this website.

Note 2: This approach is aimed to single node processes. If you want to explore (or need) distributed processing, we recommend to use the supported version, and speak with user groups and the Alliance support to help designing routines.

5.1 Introduction

In this approach, we use a prestablished RockeR, docker container from which we expand to our user needs. Binary versions of the container are available at Docker Hub, with the releases and their code available on GitHub.

5.2 Running an Apptainer HPC Container as an interactive session

Prior to be able to run RStudio, you need to have a suitable container already built as a Singularity container, or build by yourself.

From the cluster login node you can build it with the following steps (it is assumed that you are using bash as a regular user - no sudo needed):

1. Load Apptainer:

```
module load apptainer
```

Pick up the most recent version of it, and accept.

2. Then run the build (run this command at your *home*, *project* or *scratch* folder):

```
apptainer build rstudio_container_v1_0.sif docker://ricardobarroslourenco/rstudio_conta
```

You should have, at the end, a file named *rstudio_container_v1_0.sif* in the directory you ran the command. This is your Apptainer container.

3. Create transient folders and files to be able to run your container (these should be hosted at the same location as your container). Creating directories:

```
$ mkdir -p run var-lib-rstudio-server .rstudio
```

Then the sqlite configuration file:

```
printf 'provider=sqlite\ndirectory=/var/lib/rstudio-server\n' > database.conf
```

4. Starting an Interactive Job on the Supercomputer

```
$ salloc --time=0:30:0 --account=rrg-ggalex --cpus-per-task=8 --mem=0 --mail-user=macio
```

The setup of the flags is defined as this (more details available at the Alliance Documentation, as well as on the Slurm manual):

- *time*: time of the interactive session in HH:MM:SS;
- *account*: PI account at the Alliance Can project;
- *cpus-per-task*: Amount of cpus in this interactive session. Capped to the limit of the node;
- *mem*: amount of RAM you want to allocate (zero means all memory of the node);
- *mail-user*: e-mail address to send updates on the job allocation;
- *mail-type*: verbosity of the e-mail.

After running this, you probably will see your bash change to something like (in the Graham cluster):

```
(base) [lourenco@gra-login1 scratch]$ salloc --time=0:30:0 --account=rrg-ggalex --cpus
salloc: Pending job allocation 6806097
salloc: job 6806097 queued and waiting for resources
salloc: job 6806097 has been allocated resources
salloc: Granted job allocation 6806097
salloc: Waiting for resource configuration
salloc: Nodes gra189 are ready for job
(base) [lourenco@gra189 scratch]$
```

Please note that **gra189** in this case is your machine name. You need to remember it later to use it in your ssh tunnel.

5. Start the container in the newly allocated working node

```
apptainer exec --bind run:/run,var-lib-rstudio-server:/var/lib/rstudio-server,database.conf:/etc/
```

This will launch an rstudio server as a singularity service. If you need, bash is available as a terminal tab in that RStudio. To access the instance, you need to do a ssh tunnel, to forward the cluster port in the working node, to your machine. For that, **you must to create a new, separate, ssh connection, and keep it open** (without running any job on it). You can do as this (similar to what we described in the ssh section of CCPrimer, with a twist):

```
ssh -i ~/computecanada/computecanada-key -L 9102:graXXX:8787 username@graham.computecanada.ca
```

Some details:

- *-i ~/computecanada/computecanada-key*: the *-i* flag calls your ssh key file;
- *-L 9102:graXXX:8787*: the *ssh tunnel* in fact. It forwards node port 8787 to your localhost at port 9102. Note that **you must replace graXXX with the node name of the machine assigned to you** after running *salloc*;
- *username@graham.computecanada.ca*: your username, followed by the cluster url. Note that our lab research allocation is available in Graham for now.

After this, you should be able to access the rstudio server by using this url in your browser:

```
localhost:9102
```

5.3 Finishing your Running session.

Do not forget saving your work in partitions that enable saving (ex.: `/home`, `/project`, `/scratch`), because **if you save in a folder within the container, it will not be saved**, after your session ends. Remember that `/scratch` is meant for temporary save, being automatically wiped by the cluster systems after 30 days.

Once saved, you can simply end your running sessions by either using `Ctrl + C` in the apptainer run session, and then calling `exit` on bash up to ending the `salloc` session, and another time, to relinquish the ssh session (remember to do this both in the bash session that runs the jobs, as well in the ssh tunnel). At the end you will only see the bash of your local machine. It is important to do this procedure, because otherwise it may be possible that the machine is still running, and consument the group Alliance Canada credits. Once you finish the slurm `salloc` session you will receive an e-mail (as you got one when the session started), telling that the session has finished (or relinquished, if your runnign time expired).

Chapter 6

JupyterHub in Alliance Canada

Note 1: This refers to an experimental usage of Python for HPC. This approach has no official support from the Alliance, and the responsability is solely to the user. An official Python environment is provided, with support, by the Alliance at this website.

Note 2: This approach is aimed to single node processes. We will address over time cases with multi-node trainings, but this is still under development (usage of the SOSA Stack - video, slides). However, multi-node multi-gpu Deep Learning models can be already be trained with Horovod in Alliance machines (link).

6.1 Introduction

In this approach, we use an adaptation of a Docker container maintained by the Pangeo Community to install a custom setup of packages from the Conda-Forge project. Containers are available at Docker Hub, with the releases and their code available on GitHub.

6.2 Running an Apptainer HPC Container as an interactive session

Prior to be able to run JupyterLab, you need to have a suitable container already built as a Singularity container, or build by yourself.

From the cluster login node you can build it with the following steps (it is assumed that you are using bash as a regular user - no sudo needed):

1. Load Apptainer:

```
module load apptainer
```

Pick up the most recent version of it, and accept.

2. Then run the build (run this command at your *home*, *project* or *scratch* folder):

```
apptainer build ts_rs_v0_5.sif docker://ricardobarroslourenco/ts_rs:v0.5
```

You should have, at the end, a file named *ts_rs_v0_5.sif* in the directory you ran the command. This is your Apptainer container.

3. Starting an Interactive Job on the Supercomputer

```
salloc --time=24:00:0 --account=rrg-ggalex --cpus-per-task=44 --mem=0 --mail-user=maci
```

The setup of the flags is defined as this (more details available at the Alliance Documentation, as well as on the Slurm manual):

- *time*: time of the interactive session in HH:MM:SS;
- *account*: PI account at the Alliance Can project;
- *cpus-per-task*: Amount of cpus in this interactive session. Capped to the limit of the node;
- *mem*: amount of RAM you want to allocate (zero means all memory of the node);
- *mail-user*: e-mail address to send updates on the job allocation;
- *mail-type*: verbosity of the e-mail.

After running this, you probably will see your bash change to something like (in the Graham cluster):

```
(base) [lourenco@gra-login1 scratch]$ salloc --time=0:30:0 --account=rrg-ggalex --cpus
salloc: Pending job allocation 6806097
salloc: job 6806097 queued and waiting for resources
salloc: job 6806097 has been allocated resources
salloc: Granted job allocation 6806097
salloc: Waiting for resource configuration
salloc: Nodes gra189 are ready for job
(base) [lourenco@gra189 scratch]$
```

Please note that **gra189** in this case is your machine name. You need to remember it later to use it in your ssh tunnel.

4. Start the container in the newly allocated working node

```
apptainer shell --nv -B /home -B /project -B /scratch ts_rs_v0_5.sif
```

Another option, which mounts all system folders in a single *host_folders* folder can be assigned as this:

```
apptainer shell --nv -B ~/.:/host_folders/home,/home/lourenco/projects/rrg-ggalex/lourenco:/host_
```

5. Launching JupyterLab

The container is now started, and you need now to start JupyterLab. First, load the conda environment on the container:

```
source activate base
```

Note that startting Jupyter on the console will limit the scope of folders that Jupyter can see. I recommend to mount that on the root of the container filesystem. To guarantee that, let's got to the base:

```
cd /
```

Then now start a JupyterLab session (this is a session that is headless):

```
jupyter-lab --no-browser --ip $(hostname -f)
```

This will launch a JupyterLab server as a singularity service. If you need, bash is available as a terminal tab in that JupyterLab. You should see something like this:

```
Apptainer> jupyter-lab --no-browser --ip $(hostname -f)
[I 2023-06-07 11:36:57.713 ServerApp] Package jupyterlab took 0.0000s to import
[I 2023-06-07 11:36:59.595 ServerApp] Package dask_labextension took 1.8814s to import
[W 2023-06-07 11:36:59.595 ServerApp] A `_jupyter_server_extension_points` function was not found
[I 2023-06-07 11:36:59.624 ServerApp] Package jupyter_lsp took 0.0291s to import
[W 2023-06-07 11:36:59.624 ServerApp] A `_jupyter_server_extension_points` function was not found
[I 2023-06-07 11:36:59.625 ServerApp] Package jupyter_server_proxy took 0.0000s to import
[I 2023-06-07 11:36:59.644 ServerApp] Package jupyter_server_terminals took 0.0188s to import
[I 2023-06-07 11:36:59.644 ServerApp] Package notebook_shim took 0.0000s to import
[W 2023-06-07 11:36:59.644 ServerApp] A `_jupyter_server_extension_points` function was not found
[I 2023-06-07 11:37:01.808 ServerApp] Package panel.io.jupyter_server_extension took 2.1617s to i
[I 2023-06-07 11:37:01.808 ServerApp] dask_labextension | extension was successfully linked.
[I 2023-06-07 11:37:01.808 ServerApp] jupyter_lsp | extension was successfully linked.
[I 2023-06-07 11:37:01.808 ServerApp] jupyter_server_proxy | extension was successfully linked.
[I 2023-06-07 11:37:01.813 ServerApp] jupyter_server_terminals | extension was successfully linke
[I 2023-06-07 11:37:01.818 ServerApp] jupyterlab | extension was successfully linked.
[I 2023-06-07 11:37:02.927 ServerApp] notebook_shim | extension was successfully linked.
[I 2023-06-07 11:37:02.927 ServerApp] panel.io.jupyter_server_extension | extension was successfu
[I 2023-06-07 11:37:02.972 ServerApp] notebook_shim | extension was successfully loaded.
[I 2023-06-07 11:37:02.973 ServerApp] dask_labextension | extension was successfully loaded.
[I 2023-06-07 11:37:02.975 ServerApp] jupyter_lsp | extension was successfully loaded.
[I 2023-06-07 11:37:02.996 ServerApp] jupyter_server_proxy | extension was successfully loaded.
[I 2023-06-07 11:37:02.997 ServerApp] jupyter_server_terminals | extension was successfully loade
[I 2023-06-07 11:37:02.998 LabApp] JupyterLab extension loaded from /opt/conda/lib/python3.11/sit
[I 2023-06-07 11:37:02.998 LabApp] JupyterLab application directory is /opt/conda/share/jupyter/l
[I 2023-06-07 11:37:02.999 LabApp] Extension Manager is 'pypi'.
```

```
[I 2023-06-07 11:37:03.001 ServerApp] jupyterlab | extension was successfully loaded.
[I 2023-06-07 11:37:03.001 ServerApp] panel.io.jupyter_server_extension | extension was
[I 2023-06-07 11:37:03.002 ServerApp] Serving notebooks from local directory: /scratch/
[I 2023-06-07 11:37:03.002 ServerApp] Jupyter Server 2.6.0 is running at:
[I 2023-06-07 11:37:03.002 ServerApp] http://gra-login2.graham.sharcnet:8888/lab?token=
[I 2023-06-07 11:37:03.002 ServerApp]     http://127.0.0.1:8888/lab?token=4ccce5d0045f
[I 2023-06-07 11:37:03.002 ServerApp] Use Control-C to stop this server and shut down a
[C 2023-06-07 11:37:03.028 ServerApp]

    To access the server, open this file in a browser:
        file:///home/lourenco/.local/share/jupyter/runtime/jpserver-11176-open.html
    Or copy and paste one of these URLs:
        http://gra-login2.graham.sharcnet:8888/lab?token=4ccce5d0045ff5768194c8c7f37a7c
        http://127.0.0.1:8888/lab?token=4ccce5d0045ff5768194c8c7f37a7dc80220caaf68324b8
```

Note that despite several url's are provided to access the JupyterLab server, the server does not know that is running withing a container, in a cluster, therefore such URLs do not immediately work, as you are accessing it through a tunnel. The only one able to access, **after creating the ssh tunnel on port 8888**, is `http://127.0.0.1:8888/lab?token=4ccce5d0045ff5768194c8c7f37a7dc80220caaf68324b8b`.

Reinforcing, to access the instance, you need to do a ssh tunnel, to forward the cluster port in the working node, to your machine. For that, **you must to create a new, separate, ssh connection, and keep it open** (without running any job on it). You can do as this (similar to what we described in the ssh section of CCPrimer, with a twist):

```
ssh -i ~/computecanada/computecanada-key -L 8888:graXXX:8888 username@graham.computecan
```

Some details:

- *-i ~/computecanada/computecanada-key*: the *-i* flag calls your ssh key file;
- *-L 8888:graXXX:8888*: the *ssh tunnel* in fact. It forwards node port 8888 to your localhost at port 8888. Note that **you must replace graXXX with the node name of the machine assigned to you** after running *salloc*. If you use Dask in your jobs, it is available within the 8888, not needing to open aditional ports for the dashboards;
- *username@graham.computecanada.ca*: your username, followed by the cluster url. Note that our lab research allocation is available in Graham for now.

After this, you should be able to access the rstudio server by using this url in your browser:

```
http://127.0.0.1:8888/lab?token=XXXXXXXXXXXXXXXXXXXXXXX
```

Remember that this is an example, so you need to get the token URL provided by JupyterLab once launching the session.

6.3 Finishing your Running session.

Do not forget saving your work in partitions that enable saving (ex.: `/home`, `/project`, `/scratch`), because **if you save in a folder within the container, it will not be saved**, after your session ends. Remember that `/scratch` is meant for temporary save, being automatically wiped by the cluster systems after 30 days.

Once saved, you can simply end your running sessions by either using `Ctrl + C` in the apptainer run session, and then calling `exit` on bash up to ending the `salloc` session, and another time, to relinquish the ssh session (remember to do this both in the bash session that runs the jobs, as well in the ssh tunnel). At the end you will only see the bash of your local machine. It is important to do this procedure, because otherwise it may be possible that the machine is still running, and consument the group Alliance Canada credits. Once you finish the slurm `salloc` session you will receive an e-mail (as you got one when the session started), telling that the session has finished (or relinquished, if your runnign time expired).

Bibliography

Compute Canada. SSH Key — CC Doc, 2002. URL https://docs.computecanada.ca/wiki/SSH_Keys. [Online; accessed 25-February-2022].

Wikipedia contributors. Version control — Wikipedia, the free encyclopedia, 2002. URL https://en.wikipedia.org/wiki/Version_control. [Online; accessed 01-February-2022].