High Performance Computing

Contents

Today we'll be interacting with one of OSU's *High Performance Computing* clusters. HPC clusters consist of many servers (computers) that are all networked together. Each server is called a node. You can choose to work on only a single node (which will probably already be faster than your personal computer) or on several nodes at once. The point of using several nodes is that you can use them in parallel (just like we did last class when we used our own computer's cores in parallel).[1] The cluster we'll use today is called the Novus HPC cluster https://arcs.oregonstate.edu/novus-cluster that is maintained by OSU's Advance Research Computing Services.

Tools

In order to access the cluster from off-campus, you'll need to be connected to OSU's Virtual Private Network (VPN). OSU uses the *CISCO AnyConnect* client, so if you haven't already installed it, do so now. See download and instructions at https://oregonstate.teamdynamix.com/TDClient/1935/Portal/KB/ArticleDet?ID=76790. You'll need to confirm your access privileges using *Duo* and your ONID credentials.

To move files between your computer and the cluster, you'll also need an `sftp` client, like https://cyberduck.io.[2] Finally, in order to pass commands to the cluster, you'll need an `ssh` client. Mac users can use *Terminal*, which is built into the operating system. Windows users will need to install one (e.g., https://mobaxterm.mobatek.net, which also does `sftp`).

If you really don't like using the above two methods of interacting with the cluster, you could also use the `OnDemand` web interface: https://novus.dri.oregonstate.edu/.

`sftp` file transfer

Open up your `sftp` client and establish a connection to the Novus cluster using the address: `sftp://novus.dri.oregonstate.edu`. In *Cyberduck*:

1. click on "*Open Connection*",

[1] Since each node also has many cores, you can also parallelize on each of several parallel nodes, too!

[2] File transfer can also be done by `ssh` command-line, but there are no commands to remember with `sftp`.

2. select "*SFTP (Secure File Transfer Protocol)*",

3. type in the Server address:
   `sftp://novus.dri.oregonstate.edu`
   (the default *port 22* should be fine),

4. enter your ONID username and password,

5. leave *SSH Private Key* as "*None*", and

6. check the *Add to Keychain* box in order to save your credentials.[3]

You should now be in your `home` folder (e.g., `/home/novakm/`). Take a moment now to bookmark this connection. (In *Cyberduck*, select the *Add Bookmark* option from the drop-down menu.)

In your `home` folder you should see several existing folders (e.g., `ondemand`, `novus`, `globus`). There are also several "hidden" files and directories that you may or may not see depending on your *Cyberduck* settings.[4] Create a new folder **inside the `novus` folder**. Call it something informative, like the abbreviated name of your project. Open the folder and drag-and-drop your project files inside it.[5] Be sure to maintain your project's directory substructure so that all relative links between scripts and data work! In fact, you may want (or need) to put much of your whole repository in place (i.e. have your `data`, `code`, and `output` folders and contents all present).

`ssh` communication and initial `R` setup

Now switch over to your `s`ecure `sh`ell (`ssh`) client (e.g., *MobaXTerm* for Windows users, or *Terminal* for Mac users[6]). At the $ prompt, type in

`ssh` ***`yourONID`***`@novus.dri.oregonstate.edu`

[3]The option and label for this check-box may be specific to Macs.

[4]The filenames of hidden files and directories all start with a period (e.g., `.bash_profile`). Don't worry if you don't see them as you'll rarely if ever need them. In fact, you typically don't want to delete them, so be careful if you do see them.

[5]Rather than using `sftp`, you could also use `ssh` and the command
`rsync -avz ./folder_name/ ONID@novus.dri.oregonstate.edu:/home/ONID/folder_name/`
to copy from your desktop to the cluster.

[6]If you want, you can access *Terminal* from within *RStudio*. The tab for it should be in the same window as the console.

whereupon you'll be prompted for your ONID password and possibly also *Duo* confirmation. You should then see a welcome screen with a new prompt at the bottom of the window:

`[novakm@submit1 ~]$`

Since we want to run *R*, we'll have to "activate" it using the `load` command. To see what version(s) of *R* are available, and to see all the other tools ("modules") that are available as well, type

`module avail`

To load a specific module, for example *R v.4.4.1*, type:

`module load R/4.4.1`

Now launch *R* by typing-in

`R`

If you're going to need any *R* packages, you'll need to install them into your home directory using

`install.packages("`*vector_of_package_names*`")`

The first time you run the `install.packages()` command, you'll be asked if you want to create a personal library. Answer by typing in `y` for yes.
Notice that you're actually inside an *R* session! You could, therefore, work away "interactively" as if you were in an R session on your own computer. (The downside is that you don't have the benefit of writing scripts or having the ability to create graphics.)

To get out of *R* and back to the `ssh` prompt, type `q()` for quit. Don't save your workspace!

`ssh` navigation

Now lets find the files we uploaded using the `ssh` view of the cluster. Back at the `ssh` prompt, use the list (`ls`) command:

`[novakm@submit1 ~]$ ls`

The `ls` command will show you all the sub-directories and files within the directory you're currently in (which at this point in time is your `home` directory). You should see the files (or the folder) you uploaded and, if you installed any *R* packages, a folder named `R_libs`. In order to enter a subdirectory, use the change directory (`cd`) command followed by the name of the directory you'd like to enter:

`[novakm@submit1 ~]$ cd` *subdirectory*

You can move down into multiple nested dirctories, e.g.,

`[novakm@submit1 ~]$ cd` *subdirectory/subsubdirectory*
and move out of any number of directories, e.g.,
`[novakm@submit1 ~]$ cd` *../../otherdirectory*
just as we did when setting relative paths in *R*.

A very useful feature here is that you don't need to type out the whole name of a directory or file. Just type the first letter of its name and then hit your tab key. The name will autofill until it gets to a letter where it can't distinguish between similarly named files. Type the distinguishing letter and tab to continue until you've got the whole name.

Once you're in the directory in which you'd like to be, type `ls` to confirm all the contents are there as needed. If you'd like to take a quick look at the contents of a file, you can use the concatenate (`cat`) command:
`[novakm@submit1` ***subdirectory***`]$ cat` *file.r*
The contents won't be rendered very nicely, but the function is nonetheless useful for confirming the contents of short scripts (such as the `submit.sh` submission script that we'll talk about below).

`ssh` job submission (non-parallel jobs)

When performing analyses on a cluster, you (typically) don't do so be entering into the module (e.g., within *R*, the way we did above when installing *R* packages). Rather, you use a submission script to submit your code (your "job") to the cluster. The primary reason for doing that is that the cluster has extensive automated job managment tools which optimize the use of nodes among users. Therefore, when you submit a job, the cluster (typically) determines which of its nodes it will send the job to. It's similarly so when your job contains code that performs parallelized computations.

> You can find links to copies of both `submit_simple.sh` and a minimalist `simple.R` that it runs on the `README` page of today's lesson.

The submission script for a simple (non-parallel) *R* job on our cluster will look like this[7]:

[7]Note that the # symbols are *not* for commenting-out lines of code. They're necessary! (The #$, for example, denotes a command-line argument to be passed to the job scheduler.)

```
#!/bin/bash
#SBATCH -J simple
#SBATCH -o output_%A_%a.txt
#SBATCH -e errors_%A_%a.txt

#SBATCH --mail-type=ALL
#SBATCH --mail-user=ONID@oregonstate.edu

#SBATCH --partition=test.q
#SBATCH --nodes=1
#SBATCH --ntasks-per-node=1
#SBATCH --cpus-per-task=1

# Load the R Module
module load R/4.4.1

# Commands to run job
Rscript simple.R $SLURM_ARRAY_TASK_ID
```

Copy the above into a text file. Edit it as needed. At minimum, you'll want to

1. rename the `-J simple` name of the job to give it an informative project-specific name;
2. edit your email address; and
3. change the name of the *R* script at the bottom so that the appropriate *R* analysis script is called.

Save the file with an `.sh` extension (e.g., `submit_simple.sh`). Now use your `sftp` client to upload this submission script into the same directory as the `simple.R` analysis script that it calls. Ensure it's in place using your `ssh` client.

Let's assume that our analysis script and our submission script are located in a directory named `mytest`. Use `cd` to change into the directory and `ls` to ensure both your script and your submission script are present. To submit your job, use the `sbatch` command:

```
[novakm@submit1 mytest]$ sbatch submit_simple.sh
```

Job status, completion, and ending a job

You can use `squeue` to confirm that your jobs has been correctly submitted check on its status (i.e. whether it's already running, is still in the queue to be run, etc.). Use

`[novakm@submit1 mytest]$ squeue -u` *`yourONID`*

to see your jobs, and

`[novakm@submit1 mytest]$ squeue`

to see everyone else's too. Note that every job has it's own `JOBID` identifier. If your job is in the queue waiting to be run, you can use

`squeue -j <`*`JOBID`*`> --start`

to get an estimate of how long it will be before it starts running.

> Note that the `simple.R` job you submitted above will likely run so fast that it won't even be visible in your list of jobs for longer than just a few seconds. You may therefore want to submit a job that takes longer. On today's README page, you'll find a link to `simple_sleep.R` for that purpose. Remember to update your submission script.

If you typed-in your correct email address in the submission script, you'll get an email sent to you when it completes successfully or ends in an error. A log of your job will also be saved to the output file you specified in your submission script (which can be useful for debugging). The log will include output that your script printed to screen (i.e. what would have appeared in your *R* console had you run the script on your computer). Use your `sftp` client to download all the output your script produced.

Should you decide that something isn't working right for an active job (e.g., a job is taking far longer than expected, eating up too much of the cluster's resources, or you realize you do in fact have a bug in it), you can end it prematurely using the `scancel` command:

1. get its *JOBID* from the first column of the `squeue` view;
2. type `scancel` followed by the *JOBID*.

Finally, to close your `ssh` session, use the `exit` command. A list of additional commands is included on the resources page of the Novus cluster (see "Command line conversions"): https://arcs.oregonstate.edu/novus-cluster.

Parallel computing

The jobs we've just run were "simple" in the sense that we did not run anything in parallel. Rather, there was just a single script of *R* code being run on a single node of the cluster. In that sense, all "simple" jobs are the same.

Parallelization, on the other hand, can take many forms. You might want to run the same script many independent times, each time pulling in a different dataset, for example. Or your script might repeat a particular calculation many many independent times internally, as when you're doing sensitivity analyses across a `for` loop, for example. Alternatively, you might have several scripts that take different amounts of time to complete but are interdependent in that they rely on each other's output to proceed. As opposed to simple jobs, the number of possible parallel job forms is quite large. We'll only touch on examples to demonstrate the first two types.