Repo and Node

A Reproducibility Workflow + Tips and Tricks for the Cluster

Credit

Original tutorial by Michael R. May, further modified by Jenna T. B. Ekwealor

Software Prerequisites (on local machine)

Software Prerequisites (on local machine)

- **git** for data transfer (scripts and small files)
  - should come installed by default on macOS
- **GitHub Desktop** (optional but it's nice so I recommend it)

Software Prerequisites (on local machine)

- **git** for data transfer (scripts and small files)
  - should come installed by default on macOS
- **GitHub Desktop** (optional but it's nice so I recommend it)
- **homebrew** for installing software locally
  - https://brew.sh/

Software Prerequisites (on local machine)

- **git** for data transfer (scripts and small files)
  - should come installed by default on macOS
- **GitHub Desktop** (optional but it's nice so I recommend it)
- **homebrew** for installing software locally
  - https://brew.sh/
- **rsync** for data transfer (large files) between CHPC and your computer

```
brew install rsync
```

Software Prerequisites (on local machine)

- **git** for data transfer (scripts and small files)
  - should come installed by default on macOS
- **GitHub Desktop** (optional but it's nice so I recommend it)
- **homebrew** for installing software locally
  - https://brew.sh/
- **rsync** for data transfer (large files) between CHPC and your computer

```
brew install rsync
```

- **rclone** for data transfer (large files) between CHPC, BOX, and your computer

```
brew install rclone
```

Software Prerequisites (on local machine)

- **git** for data transfer (scripts and small files)
  - should come installed by default on macOS
- **GitHub Desktop** (optional but it's nice so I recommend it)
- **homebrew** for installing software locally
  - https://brew.sh/
- **rsync** for data transfer (large files) between CHPC and your computer

```
brew install rsync
```

- **rclone** for data transfer (large files) between CHPC, BOX, and your computer

```
brew install rclone
```

Introduction to CHPC

Introduction to CHPC

There are two types of things you need to worry about: **nodes** and **file systems** (hard drives).

Introduction to CHPC

Nodes are self-contained computers. They come in three varieties.

Introduction to CHPC

Login nodes (AKA **head nodes**) are where you login! This is also where you submit jobs.

NEVER run serious programs here. Even moving or copying large files can be a problem (more on that later).

Introduction to CHPC

Compute nodes are where your jobs are executed.

You should generally not log in to a compute node (unless you're doing an interactive job...). Jobs you submit on a login node are assigned to a compute node by the system.

Introduction to CHPC

The **data transfer node** is where you login when you want to move large data files to and from CHPC

I try to avoid logging into transfer nodes, and instead write scripts to move data back and forth.

Introduction to CHPC

There are two file systems (basically, sets of hard drives).

Introduction to CHPC

There are two file systems (basically, sets of hard drives).

In your *home* directory (~) you are limited to 50 GB of space, but the data are not backed up! This is the location you will want to intstall programs (if needed).

Introduction to CHPC

There are two file systems (basically, sets of hard drives).

scratch is where all of your jobs should live. Within `/scratch/` there are two file systems: * `/scratch/general/nfs1` - a 595 TB NFS system accessible from all general environment CHPC resources * `/scratch/general/vast` - 1 PB file system available from all general environment CHPC resources; quota of 50 TB per user

Each user can make their own `scratch` directory here, with essentially unlimited space (essentially!). However, there is a strict purging policy. **Files that have not been accessed for 60 days are automatically scrubbed!** These files are also not backed up.

Introduction to CHPC

There are two file systems (basically, sets of hard drives).

To get to your `scratch` directory (either in a login node, or on a compute node), use:

```
cd /scratch/general/nfs1
```

or

```
cd /scratch/general/vast
```

then create a directory for yourself with:

```
mkdir u6049165
```

Note that everyone will be able to see your directory name in the list but they won't be able to access the contents, unless you specifically give access to them.

Introduction to CHPC

Each USU faculty account can request up to 300,000 CPU (core) hours per quarter (For the "normal allocation request.") You will have to submit a request form 1-4 times per year and justify your allocation amount.

Introduction to CHPC

Each USU faculty account can request up to 300,000 CPU (core) hours per quarter (For the "normal allocation request.") You will have to submit a request form 1-4 times per year and justify your allocation amount.

It seems like a lot, but you'd be surprised how fast they disappear!

Introduction to CHPC

Each USU faculty account can request up to 300,000 CPU (core) hours per quarter (For the "normal allocation request.") You will have to submit a request form 1-4 times per year and justify your allocation amount.

It seems like a lot, but you'd be surprised how fast they disappear!

IMPORTANT: When you request a node for a job, *you are charged for all the cores on that node whether or not you use them!* We'll return to this later.

Introduction to CHPC

Each USU faculty account can request up to 300,000 CPU (core) hours per quarter (For the "normal allocation request.") You will have to submit a request form 1-4 times per year and justify your allocation amount.

It seems like a lot, but you'd be surprised how fast they disappear!

IMPORTANT: When you request a node for a job, *you are charged for all the cores on that node whether or not you use them!* We'll return to this later.

If you haven't already, find instructions to request an account and get added to Carl (or another PI's) allowance HERE.

Introduction to CHPC

CHPC uses “modules” for pre-installed programs.

Introduction to CHPC

CHPC uses "modules" for pre-installed programs.

To load a module (make the program available for use), do something like:

```
module load revbayes
```

(Rememeber to do this in any `SLURM` scripts that use the programs, too.)

Introduction to CHPC

CHPC uses "modules" for pre-installed programs.

To load a module (make the program available for use), do something like:

```
module load revbayes
```

(Rememeber to do this in any `SLURM` scripts that use the programs, too.)

You can see the list of available modules using:

```
module avail
```

You can search for a particular modlule with a keyword, for example "`iqtree`," like so:

```
module spider iqtree
```

Introduction to CHPC

- Online resources:
  - CHPC Getting Started Guide
  - Access Overview
  - Allocation Information

git, CHPC, and data transfer

git, CHPC, and data transfer

There are four systems involved.

git, CHPC, and data transfer

Two of them are “temporary” (my local machine, and CHPC)—these may get lost or deleted!

The other two are “archives” (git and BOX)—these should stay around forever!

git, CHPC, and data transfer

I use different methods to transfer small files (tracked in the repo) and large files (stored in BOX).

git, CHPC, and data transfer

I track all my small files using a git repository. Then I use push and pull (/clone) to synchronize CHPC with my local repository.

git, CHPC, and data transfer

I use `rclone` to transfer large files to CHPC. (You can also use `Globus`, `sftp`, `lftp`, `scp`, etc. I like `rclone`)

git, CHPC, and data transfer

I have one set of `rclone` scripts that transfer data from my local machine to BOX, then from BOX to CHPC.

git, CHPC, and data transfer

I have one set of `rclone` scripts that transfer data from my local machine to BOX, then from BOX to CHPC.

And another set to transfer from CHPC to BOX, then from BOX to my local machine (if necessary).

git, CHPC, and data transfer

Imagine I'm working with a git repository.

I may clone or create a repo on my local machine.

```
git clone https://github.com/jenna-tb-ekwealor/chpc_workflo
```

git, CHPC, and data transfer

I make some local changes, stage and commit them, and then push them to the remote.

```
git add .
git commit -m "some new changes!"
git pull
git push
```

git, CHPC, and data transfer

Now I want to get those changes onto CHPC

I log into CHPC, and navigate to my scratch directory

```
# log in to CHPC, for instance notchpeak
ssh u6049165@notchpeak.chpc.utah.edu

# change to my scratch directory
cd /scratch/general/nfs1/u6049165
```

git, CHPC, and data transfer

If this is the first time using the repo with CHPC, I clone the repo:

```
# clone
git clone https://github.com/jenna-tb-ekwealor/chpc_workflo

# change to that directory
cd chpc_workflow
```

git, CHPC, and data transfer

If this is the first time using the repo with CHPC, I clone the repo:

```
# clone
git clone https://github.com/jenna-tb-ekwealor/chpc_workflo

# change to that directory
cd chpc_workflow
```

Otherwise, I navigate to the repo and pull my new changes

```
# change to that directory
cd chpc_workflow

# pull
git pull
```

git, CHPC, and data transfer

But how do I transfer big data files?

git, CHPC, and data transfer

But how do I transfer big data files?

If I have a big file (e.g. a big sequence alignment), I put it somewhere in my **local** repository, but make sure to put it on my `.gitignore`!

```
# Here I am creating a fake data file,
# but imagine it's your huge genomic data
touch data/big_data.zip
```

git, CHPC, and data transfer

But how do I transfer big data files?

If I have a big file (e.g. a big sequence alignment), I put it somewhere in my **local** repository, but make sure to put it on my `.gitignore`!

```
# Here I am creating a fake data file,
# but imagine it's your huge genomic data
touch data/big_data.zip
```

Then I open my `.gitignore` file:

```
nano .gitignore
```

and add `data/big_data.zip`, then save and close. (You can use any text editor to do this.)

(Remember to stage/commit/push changes to your `.gitignore`.)

git, CHPC, and data transfer

A nice, scriptable solution for moving individual files is `rsync`.

git, CHPC, and data transfer

A nice, scriptable solution for moving individual files is `rsync`.

I'm going to use `rsync` to transfer my data file, and return to *automating* this for large data files later, when I talk about `rclone`.

git, CHPC, and data transfer

A nice, scriptable solution for moving individual files is `rsync`.

I'm going to use `rsync` to transfer my data file, and return to *automating* this for large data files later, when I talk about `rclone`.

On my **local machine**, I use `rsync` like so, to send my file to CHPC:

```
rsync -a data/big_data.zip u6049165@dtn05.chpc.utah.edu:/sc
```

(Another option is to use a file transer GUI like FileZilla or CyberDuck but we're not covering that today.)

git, CHPC, and data transfer

A nice, scriptable solution for moving individual files is `rsync`.

I'm going to use `rsync` to transfer my data file, and return to *automating* this for large data files later, when I talk about `rclone`.

On my **local machine**, I use `rsync` like so, to send my file to CHPC:

```
rsync -a data/big_data.zip u6049165@dtn05.chpc.utah.edu:/sc
```

NOTE:

- ***I'm using the data transfer node!*** See more about those here.
- -a is "archive" mode: it only updates the remote file if it's changed.
- The first argument is the *relative path to the file I want to transfer*, and the second argument is the *absolute path to the directory I want to move the file to*.

git, CHPC, and data transfer

Just to make sure the file transferred, on **CHPC**:

```
ls data/
```

Running jobs on CHPC

Now, let's submit some jobs!

Running jobs on CHPC

Now, let's submit some jobs!

CHPC uses the batch submission software `SLURM`.

Running jobs on CHPC

Now, let's submit some jobs!

CHPC uses the batch submission software `SLURM`.

`SLURM` is reponsible for coordinating everyone's jobs in the most efficient and fair way possible.

Running jobs on CHPC

Now, let's submit some jobs!

CHPC uses the batch submission software `SLURM`.

`SLURM` is reponsible for coordinating everyone's jobs in the most efficient and fair way possible.

This means your job has a certain priority based on how much time you request, and how much you have used the system recently! Sometimes you may sit in the queue for a while (in my experience, up to a day if you're doing lots of long jobs).

Running jobs on CHPC

Here is what a SLURM script looks like:

```
#!/bin/bash
#SBATCH
#SBATCH --time=00:10:00
#SBATCH --nodes=1
# additional information for allocated clusters
#SBATCH --account=rothfels
#SBATCH --partition=notchpeak
#SBATCH --mail-type=FAIL,BEGIN,END
#
#
# ----------------Modules----------------------- #
#
#
# ----------------Your Commands------------------- #
#

# change to user directory of choice
```

Running jobs on CHPC

Let's submit our job!

```
sbatch simple/simple.sh
```

Running jobs on CHPC

We can check our job(s) in the queue like so:

```
squeue -u $USER
```

(this is a short job, so don't be surprised if it's finished by the time you use `squeue`!)

Running jobs on CHPC

When you request one compute node, you get charged for all 20 cores on that node whether you use them!

The amount you get charged is nodes x cores x how long the job ran (not how much time you requested).

So, you want to either: (1) use all the cores for a given job, or (2) run multiple jobs simultaneously.

Running jobs on CHPC

Let's check out a more efficient script.

```
# run your task
echo "Task 1"  >  output/multiple.txt &
echo "Task 2"  >> output/multiple.txt &
echo "Task 3"  >> output/multiple.txt &
echo "Task 4"  >> output/multiple.txt &
echo "Task 5"  >> output/multiple.txt &
echo "Task 6"  >> output/multiple.txt &
echo "Task 7"  >> output/multiple.txt &
echo "Task 8"  >> output/multiple.txt &
echo "Task 9"  >> output/multiple.txt &
echo "Task 10" >> output/multiple.txt &
echo "Task 11" >> output/multiple.txt &
echo "Task 12" >> output/multiple.txt &
echo "Task 13" >> output/multiple.txt &
echo "Task 14" >> output/multiple.txt &
echo "Task 15" >> output/multiple.txt &
echo "Task 16" >> output/multiple.txt &
```

Running jobs on CHPC

DEMO

Running jobs on CHPC

You may notice that my `.gitignore` file ignores everything in `log` and `output` directories!

Running jobs on CHPC

You may notice that my `.gitignore` file ignores everything in `log` and `output` directories!

How am I going to get my files?!?!?!

Data transfer with `rclone`

Data transfer with `rclone`

Data transfer with `rclone`

The first thing I need to do is setup some BOX remotes.

Data transfer with `rclone`

The first thing I need to do is setup some BOX remotes.

On my **local machine**, I enter `rclone` configuration like so:

```
rclone config
```

Data transfer with `rclone`

The first thing I need to do is setup some BOX remotes.

On my **local machine**, I enter `rclone` configuration like so:

```
rclone config
```

DEMO TIME!

Data transfer with `rclone`

The first thing I need to do is setup some BOX remotes.

On my **local machine**, I enter `rclone` configuration like so:

```
rclone config
```

DEMO TIME!

I then login to the **data transfer node** and repeat.

```
# log in to CHPC DTN!
ssh u6049165@dtn05.chpc.utah.edu
```

Data transfer with `rclone`

Now, I can use `rclone` to synchronize files between storage locations.

Data transfer with `rclone`

Now, I can use `rclone` to synchronize files between storage locations.

To transfer a file from my local machine to BOX, I do this:

```
rclone sync data/ BOX_USU:chpc_workflow/data/
```

Data transfer with `rclone`

Now, I can use `rclone` to synchronize files between storage locations.

To transfer a file from my local machine to BOX, I do this:

```
rclone sync data/ BOX_USU:chpc_workflow/data/
```

The first argument (`sync`) means the remote files will only be updated if they're out of date!

The second argument (`data/`) is the file (or directories) I want to send to the remote.

The third (`BOX_USU:chpc_workflow/data/`) is the destination on BOX.

Data transfer with `rclone`

I write bash scripts to automate this procedure, so that files go into the right place every time.

Data transfer with `rclone`

I write bash scripts to automate this procedure, so that files go into the right place every time.

I also have scripts that transfer from CHPC to BOX. To use these scripts, login to the data transfer node and do something like:

```
bash synchronizers/synchronize_CHPC_to_box.sh
```

Data transfer with `rclone`

I write bash scripts to automate this procedure, so that files go into the right place every time.

I also have scripts that transfer from CHPC to BOX. To use these scripts, login to the data transfer node and do something like:

```
bash synchronizers/synchronize_CHPC_to_box.sh
```

BEWARE of relative file paths. To be consistent, I write my scripts to **run** from the top-level directory in my repository!

Thoughts

- Setting this system up can be time consuming! But I think it pays off in the end.
- It's nice to have permanent backups of your code and files. This makes your work safer and easier to share.
- This is just one example of a workflow: you could use `rclone` to transfer ALL of your files, and not use git! Or, you could use something other than `rclone` for file transfer, like `Globus` or `lftp`.

NOW FOR SOME TIPS AND TRICKS! Aliases

Aliases - Do you, too, have trouble remembering your uNID? You don't have to memorize it anymore! You can create aliases for this and other things in your cluster. Let's create a file called `.aliases` in our home directory.

```
cd ~
pwd
nano .aliases
```

This open up the text editor. Add some custom aliases like so:

```
alias jobs='squeue -u u6049165'
alias scratch='cd /scratch/general/vast/u6049165/'
```

You can do this on your local machine, too, so that you can just e.g. type `notchpeak` for `ssh u6049165@notchpeak.chpc.utah.edu`!

NOW FOR SOME TIPS AND TRICKS! Parameters file

- Say one day you're working on notchpeak, and the next day you want to re-run some scripts and notchpeak is down so you log into lonepeak instead. You'll recall your script had `#SBATCH --partition=notchpeak` at the top.
- Try running your script again and it will immediately fail! You'll have to edit it and change it to `lonepeak`. When you're running and re-running a lot of jobs, or the same jobs on different datasets, this can get tedious.
- I have partially resolved this tediousness with a `parameters.sh` file.

NOW FOR SOME TIPS AND TRICKS! Parameters file

In the same directory that you keep your jobs, create a `parameters.sh` file.

```
nano parameters.sh
```

Add this to it:

```
# Cluster settings
# Partition must be specified on each job script but may b
SBATCH_PARTITION="notchpeak"
```

From now on, in all your job scripts, start like this:

```
#!/bin/bash
# Source the parameters file
source parameters.sh # load parameters
#SBATCH --time=1:00:00
#SBATCH --nodes=1
#SBATCH --ntasks=8    # number of MPI tasks
# additional information for allocated clusters
#SBATCH --account=rothfels
```

NOW FOR SOME TIPS AND TRICKS! Parameters file

- I do the same thing for account. Though I am only using one account, I find this makes the scripts more share-able. I could now share them as is and they would not have my personal account info in them AND the user would not have to chance a million scripts to have their own account name; they'd only have to change the one parameters file.

```
# Cluster settings
# Partition must be specified on each job script but may b
SBATCH_ACCOUNT="rothfels"
```

Likewise I use `parameters.sh` for all sorts of personalization things that can be more generic in scripts. For example:

```
# Raw read filename details below
# Raw read filename extension
RAW_READS_EXT="fastq.gz"
# Raw read filename prefix
RAW_READS_PRE="RAPiD-Genomics"
# Sample prefix
```

NOW FOR SOME TIPS AND TRICKS! Parameters file

You can continue this `parameters.sh` to include any parameters for specific jobs/tools, too. e.g. I included the cleaning parameters here:

```
# Cleaning parameters (trimmomatic)
LEADING=3
TRAILING=3
SLIDINGWINDOW_LENGTH=4
SLIDINGWINDOW_SCORE=20
MINLEN=51
```

The idea is that other users (or future me) would not have to dig around into the scripts so much to make tweaks.

NOW FOR SOME TIPS AND TRICKS! Automate repetitive job submission

- Have you ever wanted to run the SAME tool on a bunch of different input files? An example that comes up for me often is building gene trees! I want all the trees to use the same `IQTREE` script, and I have 400 alignments.

One option is to run a single job that applies the `IQTREE` command to each alignment sequentially. This would *work* (and I used to do it a lot myself, so no shame!), but it's needlessly slow because there is no reason they can't run at the same time. Some tools have parallelization built in, but it's not always clear which do and which do not, and how to know.

- My favorite way to solve this problem is to run 1 job per alignment. Whoa, that's a lot of job scripts to make! Yes, it is, but we can automate it.

NOW FOR SOME TIPS AND TRICKS! Automate repetitive job submission

We are going to create some fake data files on the cluster. Make sure you are in **chpc_workflow** for this.

```
touch data/aln_{100..110}.fa
```

Use `ls data` to see what we've created!

Now let's create an empty job script on our **local machine** in the `chpc_workflow` directory.

```
nano iqtree.job
```

And here's some stuff to put at the top for now:

```
#!/bin/bash
#SBATCH --time=0:10:00
#SBATCH --nodes=1
# additional information for allocated clusters
#SBATCH --account=rothfels
#SBATCH --partition=notchpeak
#SBATCH --mail-type=FAIL,BEGIN,END
```

NOW FOR SOME TIPS AND TRICKS! Automate repetitive job submission

- This job script is not done yet, but we know we wll want to run this script for each one of the alignments in `data/`.
- Let's create a **bash** script that will submit those jobs.

```
nano submit_gene_trees.sh
```

And here is what we'll put in it:

```
# /bin/sh
mkdir -p gene_trees # make a directory for the output

for fasta in data/aln*.fa; do
    sbatch iqtree.job $fasta # submit the job for each alig
done
```

For each alignment, submit a job. Now, we have to include some special stuff in our job script!

NOW FOR SOME TIPS AND TRICKS! Automate repetitive job submission

- Return to the job script with:

```
nano iqtree.job
```

```
# build gene tree
iqtree2 -s "$1"
```

Note! instead of the actual alignment filename, I'm using "1"!*Thiswillgetreplacedbythe*`fastavariable in thesubmit_gene_trees.shjob submission script, one for each alignment (or for each version of the$fasta` variable).

Now run

```
bash submit_gene_trees.sh
```

And it will submit a job per alignment.

NOW FOR SOME TIPS AND TRICKS! Automate repetitive job submission

- This can get even fancier! You can use the command `basename` to do all sorts of renaming type stuff. For example your `submit_gene_trees.sh` could be like this:

```
# /bin/sh
mkdir -p gene_trees # make a directory for the output

for fasta in data/aln*.fa; do
    locusid=$(basename "$fasta" .fa) # grab the filename w
    echo 'the locus is' "$locusid" # for troubleshooting
    sbatch ../../scripts/_02a_build_gene_trees.job $locusi
done
```

Then your job script like so:

```
# build gene tree
iqtree2 -s "$1".faa --prefix "$1"_gene_tree
```

And so on!

THE END