# AusARG Phylogenomics Workflow

Keogh Lab: March, 2022

## Contents


**Introduction** **2**

**Data Pre-Processing** **3**
Step 0: *concatenate_collate* . . . 3

**Read Assembly** **5**
Step 1: *dedupe_clean_filter_reads* . . . 5
Step 2: *trinity_filtered_assembly* . . . 6
Step 3: *match_contigs_to_probes* . . . 7
Step 4: *make_PRG* . . . 8
Step 5: *quality_2_assembly* . . . 8

**Alignment and Gene Trees** **10**
Step 6: *phylogeny_make_alignments* . . . 10
Step 7: *phylogeny_align_genetrees* . . . 10
Step 8 (optional): *align_close_frame* . . . 11

**Crumbs** **12**
Moving Files . . . 12
Checking Usage Stats . . . 12
Downloading Containers . . . 12
No Hang Ups (nohup) . . . 13

# Introduction

This documents outlines the process of going from raw reads provided from the BioPlatforms Australia data portal through to generating gene trees. There are many ways to do each individual step in this workflow, and probably many ways it could be improved, so take notes on things that don't work as you go. Most of this is adapted from the SqCL Pipeline developed by Sonal Singhal, who has been a **huge** help throughout this process.

The working components of this workflow rely on a series of bioinformatics and phylogenetics software packages that have been *containerized* using *Singularity*. This means the packages are free standing (including dependencies), so as long as you have *Singularity v.3+* installed on your machine you can run them without any prior installation. The available packages are included in the *SqCL_Pipeline* directory that this tutorial is a part of. Have a look:

```
/SqCL_Pipeline
|-- bbmap_38.90--he522d1c_3.sif
|-- bcftools_1.12--h3f113a9_0.sif
|-- ...
|-- samtools_1.3.1--h1b8c3c0_8.sif
|-- Scripts
|   |-- align_reads1.py
|   |-- align_reads2.py
|   |-- ...
|-- trimmomatic_0.39--hdfd78af_2.sif
|-- ...
```

Parts of this workflow will take place on your local machine (downloading raw data, sample file construction), but most of it will be executed on a separate computer/server. If you're at ANU this is most likely the machine **nick** in Craig Moritz's group (*nick.rsb.anu.edu.au*). To use this server you'll need to make sure you have a login to access remotely via a terminal and are comfortable moving files back and forth with *scp*. All of the scripts you need to process your data are included in the directory *SqCL_Pipeline/Scripts*, or we will generate from some basic R functions in the file *AusARG_Tools*. Because you likely have a lot of samples to get through, we'll speed up the computing process by handling things in parallel using *GNU Parallel* to take advantage of the available resources.

# Data Pre-Processing

Let's jump in to get our data sorted and ready for processing. Let's start by opening up a new R file and sourcing the functions in *AusARG_Tools.R*

```
source("Path_to/SqCL_Pipeline/AusArg_tools.R")
```

We want to generate a sample info file that holds the basic info about each sample so we can refer to that later. The sample info file will look roughly like this:

```
## Warning in read.table(file = file, header = header, sep = sep, quote = quote, :
## incomplete final line found by readTableHeader on 'Scripts/sample_samples.csv'
```

```
##        sample
## 1 Pnig_350789
## 2 Pgra_350790
## 3 Dpax_350791
##                                                                    read1
## 1 /home/ian/SqCL_Pipeline/Optimization/Pnig_350789_R1_concat.fastq.gz
## 2 /home/ian/SqCL_Pipeline/Optimization/Pgra_350790_R1_concat.fastq.gz
## 3 /home/ian/SqCL_Pipeline/Optimization/Dpax_350791_R1_concat.fastq.gz
##                                                                    read2
## 1 /home/ian/SqCL_Pipeline/Optimization/Pnig_350789_R2_concat.fastq.gz
## 2 /home/ian/SqCL_Pipeline/Optimization/Pgra_350790_R2_concat.fastq.gz
## 3 /home/ian/SqCL_Pipeline/Optimization/Dpax_350791_R2_concat.fastq.gz
##                                                          adaptor1
## 1 GATCGGAAGAGCACACGTCTGAACTCCAGTCAC*ATCTCGTATGCCGTCTTCTGCTTG
## 2 GATCGGAAGAGCACACGTCTGAACTCCAGTCAC*ATCTCGTATGCCGTCTTCTGCTTG
## 3 GATCGGAAGAGCACACGTCTGAACTCCAGTCAC*ATCTCGTATGCCGTCTTCTGCTTG
##                                                                adaptor2   barcode1
## 1 AATGATACGGCGACCACCGAGATCTACAC*ACACTCTTTCCCTACACGACGCTCTTCCGATCT  TACGCCAAGT
## 2 AATGATACGGCGACCACCGAGATCTACAC*ACACTCTTTCCCTACACGACGCTCTTCCGATCT  TCCATGACGG
## 3 AATGATACGGCGACCACCGAGATCTACAC*ACACTCTTTCCCTACACGACGCTCTTCCGATCT  AAGTGACTTG
##     barcode2            lineage
## 1 TTCGGCTCCG  Pygopus_nigriceps
## 2 AACGTCTCCG Pletholax_gracilis
## 3 CATAAACGTC          Delma_pax
```

The file can have more columns, you can add additional information, we just won't use it in any of our scripts. But, it absolutely must ultimately have information for: *sample*, *read1*, *read2*, *adaptor1*, *adaptor2*, *barcode1*, *barcode2*, and *lineage*.

## Step 0: *concatenate_collate*

What we need as input is our library metadata file (the one you submitted with your samples for sequencing), in a .csv format and a directory where we're storing all our *fastq.gz* read files. Once we know those locations we can run the function *concatenate_collate*. What this does is (**1**) identifies all the raw read files, (**2**) extracts and combines information from the file names and the metadata file, (**3**) concatenates all forward (*R1*) and reverse (*R2*) read files for each sample separately.

```
concatenate_collate(sample.dir = "/Desktop/Optimization",
                    metadata.file = "samples_LibraryMetadata.csv",
                    outfile = "samples.csv",
                    out.path = "/home/ian/SqCL_Pipeline/Optimization/",
```

```
                    adaptor1 = "AGTCAC*ATCTC",
                    adaptor2 = "CTACAC*ACACT")
```

- **sample.dir**: full path to the folder holding your 'fastq.gz' read files
- **metadata.file**: file name for the csv metadata sequencing file (assumes it's in the 'sample.dir')
- **outfile**: name the output file
- **out.path**: this is the output directory path on the *computer you will analyze the data on* (not your local machine)
- **adaptor1** & **adaptor2**: sequence info for the adaptors, with barcodes replaced by '*'. The default value for this function will provide the appropriate adaptor sequences since they are the same for all 1536 UDIs we purchased from Perkin Elmer (adaptor+barcodes).

After executing this function, you should have successfully created a sample info file that we will use for the rest of the pipeline. One thing to consider though is how you have the *lineages* column designated. If you have multiple samples of a given species, and you want to assemble them separately, make sure to provide unique *lineage* names for each, e.g. "Homo_sapiens_1", "Homo_sapiens_2". If instead you want to assemble samples separately but collate them into a single representative of the species, you would label both the same, e.g. "Homo_sapiens".

***Troubleshooting***: This function was designed using data from our optimization experiment—and so like the rest of these scripts—has not been tested extensively on different data types. One way I can conceive of this function breaking is around concatenating the forward (R1) and reverse (R2) read files. The NextSeq500 we ran our initial sequencing on has four physical lanes per flowcell. When the BRF makes fastq files from the Illumina BCL files, it uses *bcl2fastq* which uses a lane-splitting argument that outputs four forward (L001_R1–L004_R1) and four reverse (L001_R2–L004-R2) read files, instead of the normal two (one R1, one R2). If future sequencing efforts result in a different number of output files per sample, adjust lines 59–68 in the *AusARG_Tools.R* file.

---

# Read Assembly

Almost all of the assembly and alignment scripts in this workflow can be run individually or across multiple samples in parallel. If running individually, you can check each script's commands with the *–help* flag.

```
$ python Scripts/dedupe_clean_filter_reads.py --help
```

The *–help* flag will also provide information on commands for the parallel instances too.

```
$ python Scripts/generate_shell_scripts.py --help
```

Instructions for both instances are shown below, follow along by color (individual | in parallel). To take advantage of the available computer resources, we can also generate shell scripts and run these in parallel using the single python script *generate_shell_scripts.py*. All this does is write the full python command for each sample in your *sample info* file to a file, then read it in and simultaneously process multiple commands using GNU Parallel. Most people will want to run this pipeline for many samples simultaneously.

Below, there are explanations of each script and their associated commands, as well as Suggested Usage to optimize speed and sample throughput, and Output to tell you what files/directories to expect.

## Step 1: *dedupe_clean_filter_reads*

The *dedupe_clean_filter_reads.py* script first removes identical duplicate sequences from your raw reads using *BBMap* and *dedupe.sh*. It quickly reformats the deduplicated reads back into two read files, then removes lingering adaptor and barcode sequences using *trimmomatic* and pairs up the raw reads using *pear*. Finally, it maps the reads against a reference file of phylogenetically diverse target sequences using *BBMap* and *bbmap.sh*, and removes any reads which do not match any targets with at least a user specified minimum identity threshold (*–minid*).

Suggested Usage: this step can be quite memory intensive, but you should be able to run a number of samples in parallel. Consider *–mem 10 –CPU 4 –parallel 10*. At maximum usage this will use 100Gb memory and 40 CPU.

Output: a */trim_reads* directory with *_adapters.fa, _duplicates.fq.gz, and _dd.fastq.gz* files. If filtering the reads, the final output files are *_bb.final.fastq.gz*.

Run individually:

```
~/SqCL_Pipeline$ python Scripts/dedupe_clean_filter_reads.py \
                 ---dir [...] --file [...] --mem [...] --CPU [...] \
                 --sample [...] --minid [...] --ref [...]
```

Run in parallel:

```
~/SqCL_Pipeline$ python Scripts/generate_shell_scripts.py \
                 --script dedupe_clean_filter \
                 --dir [...] \
                 --file [...] \
                 --parallel 2 \
                 --project [...] \
                 --mem 40 \
                 --CPU 40 \
                 --ref [...] \
                 --minid 0.25 \
                 --analyze yes
```

**Requirements:**

- *script*: the python script you'd like to run, here—*dedupe_clean_filter*
- *dir*: the directory path (within SqCL_Pipeline) that holds the fastq.gz files and the sample info file
- *file*: the sample info file
- *parallel*: how many instances should run simultaneously? Remember: total memory usage will be *–parallel* x *–mem*, and total CPU usage will be *–parallel* x *–CPU*
- *project*: the project name
- *mem*: memory use per instance
- *CPU*: memory use per instance
- *ref*: path to the reference fasta file of phylogenetically diverse target sequences
- *minid*: minimum identity score for bbmap.sh
- *analyze*: if not included this will just output the shell script, if *–analyze yes* will automatically run the shell script

---

## Step 2: *trinity_filtered_assembly*

The *trinity_filtered_assembly.py* script takes as input our deduplicated/cleaned/filtered reads and attempts to assemble them into contigs.

Suggested Usage: different steps in the Trinity assembler have different requirements. The first step (clustering k-mers) is highly memory intensive. The final phase (butterfly) requires high CPU capacity. Best bet is to run just a couple samples in parallel but with sufficient memory and core resources (e.g. *–mem 40 –CPU 40 –parallel 2*).

Output: *_final.fq.gz* and *_unpaired.final.fq.gz* files.

Run individually:

```
~/SqCL_Pipeline$ python Scripts/trinity_filtered_assembly.py \
                 --dir [...] --file [...] --mem [...]  --CPU [...] --sample [...]
```

Run in parallel:

```
~/SqCL_Pipeline$ python Scripts/generate_shell_scripts.py \
                 --script trinity_filtered_assembly \
                 --dir [...] \
                 --file [...] \
                 --parallel 2 \
                 --project [...] \
                 --mem 40 \
                 --CPU 40 \
                 --analyze yes
```

**Requirements:**

- *script*: the python script you'd like to run, here—*trinity_filtered_assembly*
- *dir*: the directory path (within SqCL_Pipeline) that holds the fastq.gz files and the sample info file
- *file*: the sample info file
- *parallel*: how many instances should run simultaneously? Remember: total memory usage will be *–parallel* x *–mem*, and total CPU usage will be *–parallel* x *–CPU*
- *project*: the project name
- *mem*: memory use per instance
- *CPU*: memory use per instance
- *analyze*: if not included this will just output the shell script, if *–analyze yes* will automatically run the shell script

## Step 3: *match_contigs_to_probes*

The *match_contigs_to_probes.py* script uses BLAST to match assembled contigs to our target sequences. Based on the user provided e-value (how likely the contig is to actually be the target), we can categorize each contig/target as:

- *easy_recip_match*: 1-to-1 unique match between contig and targeted locus
- *complicated_recip_match*: 1-to-1 non-unique match, in which one targeted locus matches to multiple contigs
- *ditched_no_recip_match*: a case in which the contig matches to the targeted locus, but it isn't the best match
- *ditched_no_match*: a case in which the contig matches to the targeted locus, but the locus doesn't match to the contig
- *ditched_too_many_matches*: a case in which one contig has multiple good matches to multiple targeted loci

From this, we can further choose which matches we want to keep and which to discard, but that will be in the next step.

Suggested Usage: these steps can be processed quickly for a lot of samples, consider running many samples in parallel.

Output: */matches* directory with a *blat_results* subdirectory and a *_matches.csv* file per sample.

Run individually:

```
~/SqCL_Pipeline$ python Scripts/match_contigs_to_probes.py \
                 --dir [...] --sample [...] --evalue [...] --db [...]
```

Run in parallel:

```
~/SqCL_Pipeline$ python Scripts/generate_shell_scripts.py \
                 --script match_contigs \
                 --dir [...] \
                 --file [...] \
                 --parallel 2 \
                 --project [...] \
                 --evalue [1e-30 \
                 --db [...] \
                 --analyze yes
```

**Requirements:**

- *script*: the python script you'd like to run, here—*match_contigs*
- *dir*: the directory path (within SqCL_Pipeline) that holds the fastq.gz files and the sample info file
- *file*: the sample info file
- *parallel*: how many instances should run simultaneously? Remember: total memory usage will be *–parallel* x *–mem*, and total CPU usage will be *–parallel* x *–CPU*
- *project*: the project name
- *evalue*: strictness of the contig-to-target match. More info see: What is the Expect value?
- *db*: the fasta file which holds our target sequences (imagine it as a target database). For most purposes this is *Scripts/squamate_AHE_UCE_genes_unique.flipped.fasta*
- *analyze*: if not included this will just output the shell script, if *–analyze yes* will automatically run the shell script

## Step 4: *make_PRG*

The *make_PRG.py* script allows us to choose which contig-to-probe matches we would like to keep (one of two options *easy_recip_match*, or *complicated_recip_match*), and then generates a Pseudo-Reference Genome for each sample or lineage. If you'd like it to generate each sample individually make sure the *lineage* field in your sample info file is unique for each sample. This will result in all sampled individuals being represented in the final alignments. If you'd like to generate a PRG for each "lineage" or "species", then make the *lineage* field reflect that accordingly. This will result in only one individual per "lineage" or "species" being represented in the final alignments.

Output: a */PRG* directory holding a *.fasta* file per sample.

Run individually:

```
~/SqCL_Pipeline$ python Scripts/make_PRG.py \
                 --dir [...] --file [...] --lineage [...] --keep [...]
```

Run in parallel:

```
~/SqCL_Pipeline$ python Scripts/generate_shell_scripts.py \
                 --script make_PRG \
                 --dir [...] \
                 --file [...] \
                 --parallel [...] \
                 --project [...] \
                 --keep [...] \
                 --analyze yes
```

**Requirements:**

- *script*: the python script you'd like to run, here—*make_PRG*
- *dir*: the directory path (within SqCL_Pipeline) that holds the fastq.gz files and the sample info file
- *file*: the sample info file
- *parallel*: how many instances should run simultaneously? Remember: total memory usage will be *–parallel* x *–mem*, and total CPU usage will be *–parallel* x *–CPU*
- *project*: the project name
- *keep*: which matches to keep? Probably stick with *easy_recip_match*
- *analyze*: if not included this will just output the shell script, if *–analyze yes* will automatically run the shell script

---

## Step 5: *quality_2_assembly*

The *quality_2_assembly.py* script is a quality control step that aims to give us information about the number of targets recovered per sample or lineage/species. There are two instances of it, so make sure to choose the appropriate one when identifying the *–script* command.

Output: an output directory named with the *–outdir* flag, which holds an *.assemblyquality.csv* file per sample.

Run individually:

```
~/SqCL_Pipeline$ python Scripts/make_PRG.py \
                 --dir [...] --file [...] --ind [...]
```

Run in parallel:

```
~/SqCL_Pipeline$ python Scripts/generate_shell_scripts.py \
                 --script quality_2_sample \
```

```
                --dir [...] \
                --file [...] \
                --project [...] \
                --outdir [...] \
                --analyze yes
```

**Requirements:**

- *script*: the python script you'd like to run, here—*quality_2_sample* or *quality_2_lineage*. Not sure the *lineage* implementation works right now.
- *dir*: the directory path (within SqCL_Pipeline) that holds the fastq.gz files and the sample info file
- *file*: the sample info file
- *project*: the project name
- *outdir*: where should we save the quality score files?
- *analyze*: if not included this will just output the shell script, if *–analyze yes* will automatically run the shell script

---

# Alignment and Gene Trees

## Step 6: *phylogeny_make_alignments*

The *phylogeny_make_alignments* script will run across all available samples and pull contigs into target alignments. We don't need to parallelize this, so don't worry about it. One thing to consider is the *–minsamp* flag, which determines what is the minimum number of samples required to build an alignment. Most shortcut coalescent methods like ASTRAL require quartets to determine the bipartitions, so having *–minsamp* $< 4$ is not useful.

Output: a */phylogeny* directory which holds a *locus_data.csv* summary file and a */alignments* subdirectory holding all the rough locus alignment files (*_.fasta*).

Run individually (same as in parallel):

```
~/SqCL_Pipeline$ python Scripts/phylogeny_make_alignments.py \
                 --dir [...] --file [...] --minsamp [...]
```

**Requirements:**

- *dir*: the directory path (within SqCL_Pipeline) that holds the fastq.gz files and the sample info file
- *file*: the sample info file
- *minsamp*: minimum sample number required to build an alignment (default is 4)

---

## Step 7: *phylogeny_align_genetrees*

The *phylogeny_align_genetrees_parallel* script has only a parallel implementation. It begins by finding the rough alignments, correcting the direction of contigs using MAFFT to make sure they're all consistent. It then aligns the sequences using MAFFT, and optionally trims the edges and cleans up the alignment with Gblocks. Alignments that are in the wrong direction are reversed automatically, and sequences with <100 consecutive bases are removed from alignments using bbmap. Once the alignments are clean, it builds gene trees for each alignment (if specified with *–tree_method*). If your computing resources are low, you can always run the alignment and gene tree building iteratively by setting *–CPU 1*, but expect this to proceed ***slowly***. Consensus genetrees are dropped in the */genetrees* directory.

Output: Because there are a number of steps here there are a number of intermediate files. *_.fasta* files are aligned become *_.fasta.aln* files which if trimmed become *_.fasta.aln-gb* files. Reversed alignments are noted in the *R_files.txt* file. Final alignment files are labelled *_.fasta.aln-gb.fasta. iqtree creates many intermediate files (logs etc), but consensus trees (*contree*) are located in* /phylogeny/genetrees*.

Run in parallel:

```
~/SqCL_Pipeline$ python Scripts/phylogeny_align_genetrees_parallel.py \
                 --dir [...] --trim [...] --CPU 80 --tree_method iqtree
```

**Requirements:**

- *dir*: the directory path (within SqCL_Pipeline) that holds the fastq.gz files and the sample info file
- *trim*: no commands required, but if this flag is included it will automatically clean and trim the reads. There are a number of additional parameters you can identify (*–b1*, *–b2*, *–b3*, *–b4*), but I suggest you keep them at the default settings as this will be a very light clean/trim.
- *CPU*: number of available cores, doubles as the number of parallel instances to run

---

## Step 8 (optional): *align_close_frame*

### Running in Pipeline

The *align_close_frame* script has only a parallel implementation. It takes our AHE loci (coding sequences) and uses MACSE to set the alignment into an appropriate reading frame so that the first basepair is in reading frame 1, and the last reading frame is complete. This means that we can concatenate all AHE loci and then partition each base to the appropriate codon position. E.g. basepairs 1,4,7,10...100,103...1000,1003 = codon 1. Basepairs 2,5,8...101,104... = codon 2. Basepairs 3,6,9...103,106... = codon 3. MACSE works by translating the nucleotide alignment into an amino acid alignment. There are a few options for the program, but probably best to just stick to the *refine* option.

Output: *_NT.fasta* and *_AA.fasta* alignment files are located in respective subdirectories of */macse*.

Run in parallel:

```
~SqCL_Pipeline$ python Scripts/align_close_frame.py \
                --dir [...] --program [...] --CPU 40 --trimmed
```

**Requirements:**

- *dir*: the directory path (within SqCL_Pipeline) that holds the fastq.gz files and the sample info file
- *program*: options are "refine", "refineLemmon", "export", "align". *refine* and *refineLemmon* work with a good existing alignment and just tweaks it slightly, and so is comparatively fast. *align* assumes the input alignment is raw and needs considerable work, and so is much slower. *export* quickly reads in the file, and spits out a version with closed reading frames, but does little else.
- *CPU*: number of available cores, doubles as the number of parallel instances to run
- *trimmed*: use the *–trimmed* flag if you previously trimmed your alignments using Gblocks. Otherwise it assumes the alignments files do not include the "-gb" suffix.

### Running out of Pipeline

If you'd like to operate this script outside the pipeline, you can use the generalized version *align_close_frame_general.py*. There are just a few key differences.

Output: *_NT.fasta* and *_AA.fasta* alignment files are located in respective subdirectories of */macse*.

Run in parallel:

```
~SqCL_Pipeline$ python Scripts/align_close_frame_general.py \
                --dir [...] --indir [...] --program [...] --CPU 40
```

**Requirements:**

- *dir*: the directory path (within SqCL_Pipeline) that holds the fastq.gz files and the sample info file
- *indir*: directory that holds the alignment files. If you input *–dir frogs/*, then *–indir* will be *–indir frogs/[alignment dir]*
- *program*: options are "refine", "refineLemmon", "export", "align". *refine* and *refineLemmon* work with a good existing alignment and just tweaks it slightly, and so is comparatively fast. *align* assumes the input alignment is raw and needs considerable work, and so is much slower. *export* quickly reads in the file, and spits out a version with closed reading frames, but does little else.
- *CPU*: number of available cores, doubles as the number of parallel instances to run

# Crumbs

Some additional resources which might be useful.

---

## Moving Files

It can be a nuisance to move files between your local computer and another machine. Use *scp* to copy files using your terminal.

```
local_machine$ scp [path_on_local] [path_on_server]
```

For example, if we wanted to move the file *geckos.fasta* from our local machine to our home directory on the server **nick**:

```
local_machine$ scp path_to/geckos.fasta ian@nick.rsb.anu.edu.au:/home/ian
```

Alternatively when we want to move an output file (*updated_geckos.fasta*) from the server back to our local machine:

```
local_machine$ scp ian@nick.rsb.anu.edu.au:/home/ian/path_to/updated_geckos.fasta [local_path_destinati
```

Sometimes we need to move a folder or a series of files with similar extensions.
To move an entire directory:

```
local_machine$ scp -r [path_to_directory] [path_to_destination]
```

To move a series of fasta files:

```
local_machine$ scp path_to/*.fasta [path_to_destination]
```

---

## Checking Usage Stats

Before running any analyses it's important to make sure there are sufficient available computing resources (e.g. someone else isn't hogging the server).
Quickly check availability with *htop*.

```
server$ htop
```

This will show the number of available CPUs, amount of memory being used, who is using it, et al.
If we're only interested in the process that our account is running:

```
server$ htop -u [username]
```

We can also quickly check available disk space with *free*.

```
server$ free -h
```

This will show the amount of hard disk space installed, available, and in use.

## Downloading Containers

The included containerized packages have the file extension *.sif* (Singularity Image File) and have been downloaded from *quay.io using *Singularity*. E.g.

```
$ singularity pull docker://quay.io/biocontainers/mafft:7.480--h779adbc_0
```

There are *tons* of other available packages/programs, so have a browse.

## No Hang Ups (nohup)

The steps in this workflow will take varying amounts of time, from an instant to ages. For the longer steps (*dedupe_clean_filter_reads.py*, *phylogeny_align_genetrees_parallel.py*) you may encounter an error as a result of your shell disconnecting from the server: `client_loop: send disconnect: Broken pipe`. To get around this you can use `nohup` to keep the process running without hanging up, even if you get disconnected. The basic syntax is:

```
server$ nohup [comand you want to run] > std.out &
```

Basically this says, don't hangup (`nohup`), run this command (`[commands]`), write output to (`>`), this document (`std.out`), and keep it running in the background (`&`).
If I wanted to run the alignment and genetree generating step (which may take ~1-2 days), I'd run:

```
server$ nohup python Scripts/phylogeny_align_genetrees_parallel.py \
--dir Anilios --trim --CPU 80 --tree_method iqtree > std.out &
```

The