Reproductive research with Snakemake and Git

Introduction

In this tutorial you are going to learn the basic concepts of building workflows with Snakemake. You are also going to learn how to use Git to control your code and Bitbucket to keep it always safe of accidental deletions. To achieve this goal, we are going to replicate part of the RNA-seq workflow published by Pertea et al, 2016.

Preparing the environment and adding our first file to Git version control

To begin, let's create a directory called *my_rnaseq_exp* and move to it. This directory name is expected for some of the scripts published with the paper. Therefore, let's keep this name.

The following command lines are going to create the new directory and move to it.

```
mkdir my_rnaseq_exp
cd my_rnaseq_exp
```

To run Snakemake, we need to create a file called *Snakefile*, that will contain the rules to be executed. Let's take advantage of this step to put in practice some concepts of Git and learn how to connect our code with a Bitbucket repository. Therefore, before creating any file, let's create a Git online repository and connect it to our local directory.

Access your Bitbucket account now. If you haven't one account yet, create one at bitbucket.org.

In the left side of the screen, click on the plus sign `(+)`, then on 'Repository'. Create a name for both your project and your repository. For now, let's skip the creation of a README file by selecting **'No'** in the option `'include a README'`.

Make sure to select the box called 'Private repository'! It will insure that your files are not visible on internet.

After creating the repository, you will see a page with instructions on how to connect the online repository to your local directory. Before executing these commands, we need to initiate Git on our directory. You will need to run the following command line:

```
git init
```

Now we can connect with our Bitbucket repository. Execute the first line of **step2** shown in the Bitbucket webpage. It should look like the line below, but should contain the information of your user and repository names.

```
git remote add origin
https://Wendell_Pereira@bitbucket.org/Wendell_Pereira/psc_workshop.git
```

The repository is now connected to our local directory, but we do not have any files under version control yet. Let's create a file to insert the Snakemake rules and add it to version control by Git. In the command lines below, `touch` creates a new, empty file called Snakefile.

```
touch Snakefile
```

Now, lets add it to Git:

```
git add Snakefile
```

Using a text editor, like nano or VI, let's insert a short comment in the file. I suggest a short description as a title or any other information of your choice. The goal here is just to make changes in the content of the file.

Since we want Snakemake to understand it as a comment, we need to start each line with '#'.

For example, I am going to add this header:

```
# May 15th, PSC Bioinformatics Workshop - Snakemake Workflows
# Wendell Pereira (wendellpereira@ulf.edu)
```

Now, let's tell Git to record the changes, and create our first commit.

```
git commit -m "My first commit!" Snakefile
```

Git stores all your commits in a hidden folder insider your current directory. You can see the folder using `ls -lha` in the terminal. In general, this folder is protected from accidental deletions. However, they can happen and it is always preferable to synchronize our files and commits with an online repository, to keep them safe. Since we already connected our local directory with the online version on Bitbucket, we can just transfer the information using the `git push` command, as shown below.

```
git push origin master
```

Since your repository is private, Bitbucket will ask your password in the terminal.

We don't need to go deep in the details of how Git controls the branches, and into the meaning of the nomenclatures like origin and master. For now, we just need to know that `git push` is sending the information from our local machine to the online repository. In the same way, `git pull` will download the information of the online directory to the local one.

For more information, you can check the Git manual here. Also, UF Research Computing team frequently offers training on Git and Github (which works similar to Bitbucket). There are also multiple sources, including many blog posts and videos, on how to use Git on the internet.

Now, you can see that the Snakefile is in your online repository. To see how `git pull` works, let's add a README file in our online repository. Click on ⋯ on the upper-right corner of the Bitbucket webpage, then on `add a file`. Type 'README' as the file name and type a description of your choice. Finally, click on commit.

For example:

```
This repository contains the contents of the "PSC Bioinformatics Workshop - Snakemake
Workflows", May 15th.
```

Now, we have a file that only exists in the online repository. To download it for your local repository, use the follow command:

```
git pull origin master
```

Now, we should see a README file in our local repository. This is a very simple example, but you can use Git to synchronize files on multiple computers, i.e. on HiperGator and on your local computer. Moreover, multiple people can work in the same repository, though the process of synchronizing the changes is a little more complex than what was shown here.

Be aware of conflicts between versions in the online and local repositories. For example, if you altered a file in the online repo and also changed it in the local version, Git will not be able to synchronize it easily. Instead, it will offer you an option of merging the different versions, which may cause unintended changes. Try to always keep the versions updated, synchronizing the files before you work on them on a different machine or repository.

Constructing the pipeline

Now that we have our Snakefile, let's start to create our pipeline by adding rules that are going to execute the necessary tasks to replicate part of the results described on Pertea et al, 2016.

But first, we need to download the input data. The list of files available to download can be found here.

Download of the input data

The following steps are going to download and prepare the datasets described on Pertea et al, 2016 to be used. Note that we first create a 'developmental Slurm session' to avoid executing commands in the login servers of HiperGator.

```
module load ufrc
srundev --time=2:00:00 --ntasks=1 --cpus-per-task=1 --mem=2gb

# Download
wget ftp://ftp.ccb.jhu.edu/pub/RNAseq_protocol/chrX_data.tar.gz
tar xvzf chrX_data.tar.gz

# Removes the compresssed file
rm chrX_data.tar.gz
```

In general, we do not include the download of inputs in the workflow. Instead, the pipeline is usually constructed to access the data in the current, or in a specific directory.

Adding our first rule - mapping the reads

Initially, let's keep the code as simple as possible. Then, we will start to modify it to make it more flexible. Below is the code of a rule that will execute the mapping of the reads.

```
rule mapping_opt1:
    input:
        "chrX_data/samples/ERR188044_chrX_1.fastq.gz",
        "chrX_data/samples/ERR188044_chrX_2.fastq.gz",
        "chrX_data/samples/ERR188104_chrX_1.fastq.gz",
        "chrX_data/samples/ERR188104_chrX_2.fastq.gz",
        "chrX_data/samples/ERR188234_chrX_1.fastq.gz",
        "chrX_data/samples/ERR188234_chrX_2.fastq.gz",
        "chrX_data/samples/ERR188245_chrX_1.fastq.gz",
        "chrX_data/samples/ERR188245_chrX_2.fastq.gz",
        "chrX_data/samples/ERR188257_chrX_1.fastq.gz",
        "chrX_data/samples/ERR188257_chrX_2.fastq.gz",
        "chrX_data/samples/ERR188273_chrX_1.fastq.gz",
        "chrX_data/samples/ERR188273_chrX_2.fastq.gz",
        "chrX_data/samples/ERR188337_chrX_1.fastq.gz",
        "chrX_data/samples/ERR188337_chrX_2.fastq.gz",
        "chrX_data/samples/ERR188383_chrX_1.fastq.gz",
        "chrX_data/samples/ERR188383_chrX_2.fastq.gz",
        "chrX_data/samples/ERR188401_chrX_1.fastq.gz",
        "chrX_data/samples/ERR188401_chrX_2.fastq.gz",
        "chrX_data/samples/ERR188428_chrX_1.fastq.gz",
        "chrX_data/samples/ERR188428_chrX_2.fastq.gz",
```

```
        "chrX_data/samples/ERR188454_chrX_1.fastq.gz",
        "chrX_data/samples/ERR188454_chrX_2.fastq.gz",
        "chrX_data/samples/ERR204916_chrX_1.fastq.gz",
        "chrX_data/samples/ERR204916_chrX_2.fastq.gz"
    output:
        "ERR188044_chrX.sam",
        "ERR188104_chrX.sam",
        "ERR188234_chrX.sam",
        "ERR188245_chrX.sam",
        "ERR188257_chrX.sam",
        "ERR188273_chrX.sam",
        "ERR188337_chrX.sam",
        "ERR188383_chrX.sam",
        "ERR188401_chrX.sam",
        "ERR188428_chrX.sam",
        "ERR188454_chrX.sam",
        "ERR204916_chrX.sam"
    shell:
        """
        hisat2 -p 6 --upto 200000 --dta -x chrX_data/indexes/chrX_tran -1
chrX_data/samples/ERR188044_chrX_1.fastq.gz -2
chrX_data/samples/ERR188044_chrX_2.fastq.gz -S ERR188044_chrX.sam
        hisat2 -p 6 --upto 200000 --dta -x chrX_data/indexes/chrX_tran -1
chrX_data/samples/ERR188104_chrX_1.fastq.gz -2
chrX_data/samples/ERR188104_chrX_2.fastq.gz -S ERR188104_chrX.sam
        hisat2 -p 6 --upto 200000 --dta -x chrX_data/indexes/chrX_tran -1
chrX_data/samples/ERR188234_chrX_1.fastq.gz -2
chrX_data/samples/ERR188234_chrX_2.fastq.gz -S ERR188234_chrX.sam
        hisat2 -p 6 --upto 200000 --dta -x chrX_data/indexes/chrX_tran -1
chrX_data/samples/ERR188245_chrX_1.fastq.gz -2
chrX_data/samples/ERR188245_chrX_2.fastq.gz -S ERR188245_chrX.sam
        hisat2 -p 6 --upto 200000 --dta -x chrX_data/indexes/chrX_tran -1
chrX_data/samples/ERR188257_chrX_1.fastq.gz -2
chrX_data/samples/ERR188257_chrX_2.fastq.gz -S ERR188257_chrX.sam
        hisat2 -p 6 --upto 200000 --dta -x chrX_data/indexes/chrX_tran -1
chrX_data/samples/ERR188273_chrX_1.fastq.gz -2
chrX_data/samples/ERR188273_chrX_2.fastq.gz -S ERR188273_chrX.sam
        hisat2 -p 6 --upto 200000 --dta -x chrX_data/indexes/chrX_tran -1
chrX_data/samples/ERR188337_chrX_1.fastq.gz -2
chrX_data/samples/ERR188337_chrX_2.fastq.gz -S ERR188337_chrX.sam
        hisat2 -p 6 --upto 200000 --dta -x chrX_data/indexes/chrX_tran -1
chrX_data/samples/ERR188383_chrX_1.fastq.gz -2
chrX_data/samples/ERR188383_chrX_2.fastq.gz -S ERR188383_chrX.sam
        hisat2 -p 6 --upto 200000 --dta -x chrX_data/indexes/chrX_tran -1
chrX_data/samples/ERR188401_chrX_1.fastq.gz -2
chrX_data/samples/ERR188401_chrX_2.fastq.gz -S ERR188401_chrX.sam
        hisat2 -p 6 --upto 200000 --dta -x chrX_data/indexes/chrX_tran -1
chrX_data/samples/ERR188428_chrX_1.fastq.gz -2
chrX_data/samples/ERR188428_chrX_2.fastq.gz -S ERR188428_chrX.sam
        hisat2 -p 6 --upto 200000 --dta -x chrX_data/indexes/chrX_tran -1
chrX_data/samples/ERR188454_chrX_1.fastq.gz -2
```

```
chrX_data/samples/ERR188454_chrX_2.fastq.gz -S ERR188454_chrX.sam
        hisat2 -p 6 --upto 200000 --dta -x chrX_data/indexes/chrX_tran -1
chrX_data/samples/ERR204916_chrX_1.fastq.gz -2
chrX_data/samples/ERR204916_chrX_2.fastq.gz -S ERR204916_chrX.sam
        """
```

A common error is to forget the comma between the input or output items. Snakemake will return an error that the syntax is wrong.

Copy the above commands to your Snakefile. Now, to visualize what Snakemake will execute, we can run a 'dry-run' (using option `-n`). This will show what will be executed, without actually running the task. We will also add the option `-p` to print the shell code.

If you are running it on HiperGator, you need to load the Snakemake module first. If you are running in your computer, skip the command to load the module.

For HiperGator only!

```
module load snakemake/5.17.0
```

```
#snakemake -n -p mapping_opt1
snakemake -np mapping_opt1
```

It is possible to execute the command without adding the name of the rule (`snakemake -np`). When we do that, Snakemake will try to execute the first rule in the Snakefile. Since we only have one, the final result will be the same. Later, we will take advantage of this aspect to guarantee the execution of all rules in the pipeline.

Some points to consider:

- In the mapping rule, we still write all file names and parameters in the rule itself. Therefore, this rule is not flexible. That means you CANNOT use it with different input files.
- The code is structured by the insertion of indentations. Snakemake relies on this structure and the code will fail if it is not respected.
- The code in the shell block is equivalent to executing line by line in a terminal.

As we can see, it is a long command and there is no advantage of using this structure instead of a shell script yet. For that, we need to start to take advantage of the Snakemake functions. Let's modify the code to a structure more similar to what we would do in real life.

```
rule mapping_opt2:
    input:
        "chrX_data/samples/ERR188044_chrX_1.fastq.gz",
        "chrX_data/samples/ERR188044_chrX_2.fastq.gz",
        "chrX_data/samples/ERR188104_chrX_1.fastq.gz",
```

```
        "chrX_data/samples/ERR188104_chrX_2.fastq.gz",
        "chrX_data/samples/ERR188234_chrX_1.fastq.gz",
        "chrX_data/samples/ERR188234_chrX_2.fastq.gz",
        "chrX_data/samples/ERR188245_chrX_1.fastq.gz",
        "chrX_data/samples/ERR188245_chrX_2.fastq.gz",
        "chrX_data/samples/ERR188257_chrX_1.fastq.gz",
        "chrX_data/samples/ERR188257_chrX_2.fastq.gz",
        "chrX_data/samples/ERR188273_chrX_1.fastq.gz",
        "chrX_data/samples/ERR188273_chrX_2.fastq.gz",
        "chrX_data/samples/ERR188337_chrX_1.fastq.gz",
        "chrX_data/samples/ERR188337_chrX_2.fastq.gz",
        "chrX_data/samples/ERR188383_chrX_1.fastq.gz",
        "chrX_data/samples/ERR188383_chrX_2.fastq.gz",
        "chrX_data/samples/ERR188401_chrX_1.fastq.gz",
        "chrX_data/samples/ERR188401_chrX_2.fastq.gz",
        "chrX_data/samples/ERR188428_chrX_1.fastq.gz",
        "chrX_data/samples/ERR188428_chrX_2.fastq.gz",
        "chrX_data/samples/ERR188454_chrX_1.fastq.gz",
        "chrX_data/samples/ERR188454_chrX_2.fastq.gz",
        "chrX_data/samples/ERR204916_chrX_1.fastq.gz",
        "chrX_data/samples/ERR204916_chrX_2.fastq.gz"
    params:
        threads=6,
        n_of_reads=200000,
        genome_index="chrX_data/indexes/chrX_tran"
    output:
        "ERR188044_chrX.sam",
        "ERR188104_chrX.sam",
        "ERR188234_chrX.sam",
        "ERR188245_chrX.sam",
        "ERR188257_chrX.sam",
        "ERR188273_chrX.sam",
        "ERR188337_chrX.sam",
        "ERR188383_chrX.sam",
        "ERR188401_chrX.sam",
        "ERR188428_chrX.sam",
        "ERR188454_chrX.sam",
        "ERR204916_chrX.sam"
    shell:
        "hisat2 -p {params.threads} --upto {params.n_of_reads} --dta -x
{params.genome_index} -1 {input[0]} -2 {input[1]} -S {output[0]};"
        "hisat2 -p {params.threads} --upto {params.n_of_reads} --dta -x
{params.genome_index} -1 {input[2]} -2 {input[3]} -S {output[1]};"
        "hisat2 -p {params.threads} --upto {params.n_of_reads} --dta -x
{params.genome_index} -1 {input[4]} -2 {input[5]} -S {output[2]};"
        "hisat2 -p {params.threads} --upto {params.n_of_reads} --dta -x
{params.genome_index} -1 {input[6]} -2 {input[7]} -S {output[3]};"
        "hisat2 -p {params.threads} --upto {params.n_of_reads} --dta -x
{params.genome_index} -1 {input[8]} -2 {input[9]} -S {output[4]};"
        "hisat2 -p {params.threads} --upto {params.n_of_reads} --dta -x
{params.genome_index} -1 {input[10]} -2 {input[11]} -S {output[5]};"
```

```
        "hisat2 -p {params.threads} --upto {params.n_of_reads} --dta -x
{params.genome_index} -1 {input[12]} -2 {input[13]} -S {output[6]};"
        "hisat2 -p {params.threads} --upto {params.n_of_reads} --dta -x
{params.genome_index} -1 {input[14]} -2 {input[15]} -S {output[7]};"
        "hisat2 -p {params.threads} --upto {params.n_of_reads} --dta -x
{params.genome_index} -1 {input[15]} -2 {input[17]} -S {output[8]};"
        "hisat2 -p {params.threads} --upto {params.n_of_reads} --dta -x
{params.genome_index} -1 {input[16]} -2 {input[19]} -S {output[9]};"
        "hisat2 -p {params.threads} --upto {params.n_of_reads} --dta -x
{params.genome_index} -1 {input[17]} -2 {input[21]} -S {output[10]};"
        "hisat2 -p {params.threads} --upto {params.n_of_reads} --dta -x
{params.genome_index} -1 {input[18]} -2 {input[23]} -S {output[11]};"
```

Basically, what we did was use the variables `input`, `output` and `params` as substitutes for the real file names or parameter values. Nonetheless, we still need to add all this information in the rule itself, which makes it still not flexible. To verify how the shell commands are going to be executed, let's execute a new dry-run.

```
snakemake -np mapping_opt2
```

Now that we have seen how to use these variables as part of our code, let's go through the main step to make it flexible: using a configuration file. In the next section, we will create flexible rules that could be used in a real-life workflow.

However, before that, let's save the changes in our code using git commit. No need to synchronize it with the online repository yet.

```
git commit -m "Add two options of rules to execute the mapping of the reads in the
reference genome." Snakefile
```

Flexibility in the workflow through using a configuration file

A configuration file can be write in JSON or YAML. We will use YAML because, in my opinion, it produces a file that's easier to read.

To begin, let's create a new file and add it to Git.

```
touch config.yaml
git add config.yaml
```

Now, we need to define the connection between "Snakefile" and "config.yaml". It is very simple to do. We just need to add the line of code below in the beginning of the Snakefile (right after the comments).

```
configfile: "config.yaml"
```

The `configfile:` command tells Snakemake to access the parameters from the configuration file.

For the next step, we will use the `expand()` function to help us to generate the input and output file names. In addition, all parameters are going to be read from the config.yaml file.

In the Snakefile, add the following command lines. The code seems complicated at the first. However, it is quite simple once you understand how the information from the configuration file is used. We will go though all aspects and **don't be shy about using the Zoom chat if you have any questions!**

```
rule mapping:
    input:
        expand("{dir}/{samples}_{inputs_sufix}",
        dir=config["mapping"]["inputs_directory"],
        samples=config["mapping"]["inputs"],
        inputs_sufix=config["mapping"]["inputs_sufix"]),
    params:
        genome_index=config["mapping"]["params"]["genome_index"],
        threads=config["mapping"]["params"]["threads"],
        n_of_reads=config["mapping"]["params"]["n_of_reads"],
        samples=config["mapping"]["inputs"],
        inputs_sufix=config["mapping"]["inputs_sufix"],
        dir=config["mapping"]["inputs_directory"],
    output:
        expand("mapping_out/{samples}.sam", samples=config["mapping"]["inputs"])
    shell:
        """
        for file in {params.samples};
        do

            hisat2 -p {params.threads} --upto {params.n_of_reads} --dta -x
{params.genome_index} -1 {params.dir}/${{file}}_{params.inputs_sufix[0]} -2
{params.dir}/${{file}}_{params.inputs_sufix[1]} -S mapping_out/${{file}}.sam

        done
        """
```

The use of a for loop in the shell block of code allows us to work with different numbers of samples. Since the same operation needs to be executed for each sample, and the for loop will process all of them, at the end, we have one output file for each sample.

We directed the output files to the directory `mapping_out`. However, we did not create this directory yet. Snakemake will generated it automatically!

Now, let's add the information of our samples in the configuration file. Copy and paste the following code in the config.yaml file.

```
mapping:
    params:
        genome_index: chrX_data/indexes/chrX_tran
        threads: 6
        n_of_reads: 200000
    inputs:
        [
    "ERR188044_chrX",
    "ERR188104_chrX",
    "ERR188234_chrX",
    "ERR188245_chrX",
    "ERR188257_chrX",
    "ERR188273_chrX",
    "ERR188337_chrX",
    "ERR188383_chrX",
    "ERR188401_chrX",
    "ERR188428_chrX",
    "ERR188454_chrX",
    "ERR204916_chrX",
        ]
    inputs_sufix: ["1.fastq.gz", "2.fastq.gz"]
    inputs_directory: "chrX_data/samples"
```

Similar to Snakefile, the configuration file also accepts comments starting with "#".

The main point here is that `config` stores the information present in the config.yaml. Therefore, all we need to do is use `["name"]` to point to the variables in the 'config.yaml' file. If there are multiple nested levels, we need to use multiple brackets. For example, `["mapping"]["params"]["threads"]` recovers the number `6`, since it is stored in `mapping: params: threads:` entrance of the 'config.yaml' file.

To help us to understand how the rule works, lets execute a dry-run using `-n`. We also want to visualize the shell commands. For that, we can add `-p` (`-np` to combine both).

```
snakemake -np mapping
```

If everything worked well, we can see that the function `expand()` generates the same inputs and outputs as listed in the previous rules. However, now it will work for any number of samples that are listed in the configuration file. Since we also used a for loop, the shell part of the code is also independent of the number of samples. Therefore, we can consider this rule as a flexible one, since it can deal with any number of samples that have the same characteristics of the samples we are using.

So far, we have the rule we intend to execute, but we did not run any of them. For now, you will trust that all rules produce the same results (You can run them by yourself later to prove it, if you

wish) and will keep only the option 3 in our Snakefile.

Delete the previous rules from your Snakefile, add the rule mapping and save it. Also, use `git commit` to track this change and `git push` to syncronize to Bitbucket. Let's check in our online repository to see what the code looks like and how all the commits are registered — even the one we did not synchronize before.

```
git commit -m "Substitutes the previous mapping rules for one improved version"
Snakefile
git commit -m "Adds the parameters for the mapping rule" config.yaml

git push origin master
```

Finally, let's execute the code. Since we want to use multiple cores, we need to start a new developmental session on HiperGator. If you are running it on your computer, you can skip this step and run only the snakemake command.

For HiperGator only!

```
# exit the previous session and starts a new one
exit
srundev --time=2:00:00 --ntasks=1 --cpus-per-task=6 --mem=10gb

# Load snakemake and
module load snakemake/5.17.0
module load hisat2/2.2.0
```

```
snakemake -p mapping
```

While running the mapping rule above, we did not inform snakemake about the number of available cores. It will recognize how many cores we have and use as many as the rule requests.

If the rule request more cores than we have available, snakemake will reduce the number of cores of the rule to make it compatible. If we have more cores that we need, snakemake may run multiple jobs of a pipeline at the same time.

If you are on HiperGator, it will run, but you will notice an error message. That is because Snakemake uses python3 and Hisat loads python2, generating a conflict.

Be aware of dependencies conflicts in HiperGator! Sometimes, we need to load a module right before the rule that depends on it, then remove it and load other modules that following rules depend on, or reload the modules that were broken by the conflict.

Most of the time, we do not want to run our jobs in the developmental session of HiperGator. Instead, we need to submit a job to Slurm scheduler (using a sbatch script, for example). It is

possible to do that using Snakemake too, as we will learn in the next section.

An example of a sbatch script

```
#!/bin/sh
#SBATCH --job-name=prank                    # Job name
#SBATCH --mail-type=ALL                     # Mail events (NONE, BEGIN, END, FAIL,
ALL)
#SBATCH --mail-user=wendellpereira@ufl.edu  # Where to send mail
#SBATCH --nodes=1                           # Use one node
#SBATCH --ntasks=1                          # Run a single task
#SBATCH --cpus-per-task=1                   # Number of CPU cores per task
#SBATCH --mem-per-cpu=10gb                  # Memory per processor
#SBATCH --time=96:00:00                     # Time limit hrs:min:sec
#SBATCH --qos=kirst                         # qos
#SBATCH --output=prank_%A-%a.out            # Standard output and error log

# Load required modules
module load cnspipeline/20170827
module load maffilter/1.3.0
module load R/3.6
module load bedtools/2.27.1
module load perl/5.24.1
module load prank/170427
module load phast/1.5

#Command lines
```

Running Snakemake in a Slurm environment

To submit jobs in Slurm using Snakemake, we need to incorporate the `--cluster` option during the execution. This will store the parameters that Slurm requires. If you are used to running sbatch scripts, `--cluster` is the place where you need to insert the information that goes in the header of an sbatch script.

Similarly to what we do when running sbatch scripts, we also need to load the modules that are going to be executed during the Slurm job. The easiest way to do it is by adding some Python code in the Snakefile that will tell Snakemake to load the modules before executing the rules.

If you are running it on your computer, you can skip this step.

If you are running it on HiperGator, add these lines right before the first rule.

```
run_in_slurm_env = True
if run_in_slurm_env == True:
    shell.prefix("module load hisat2/2.2.0; module load stringtie/2.0.4; module load
samtools/1.3.1; module load stringtie/2.0.4; module load gffread/0.9.8c")
```

In this way, Snakemake will be able to use these parameters to submit your job as a shell script. Here we are going to execute only one rule. However, in real life projects, you can submit your whole workflow and Snakemake will manage the jobs submission for you.

It is easier to understand using an example. Let's check the command to execute the mapping rule, described above, as a slurm job.

```
snakemake -np -j 10 --cluster "sbatch -A kirst --mem=4gb -t 02:00:00 -c 6 -o
mapping_%j.out -e mapping_%j.err --mail-user wendellpereira@ufl.edu --mail-type ALL"
mapping
```

Let's understand the parameters in this command line:

The following are **Snakemake parameters**:

- `-n` Runs a dry-run (not execute the commands)
- `-p` Print the shell commands
- `-j` Controls the maximum number of jobs that Snakemake can have submitted at a time. That is useful so you can guarantee that you are not taking all resources from your qos.
- `--cluster` This is the submission command that should be used for the scheduler (in our case, Slurm). Note that command flags which normally are put in batch scripts are put here.

The parameters below are **Slurm options** and must be given for the `--cluster` option.

- `-A` Account name (or the qos that contains the resources you can access)
- `--mem` Amount of memory
- `-t` Requested time
- `-c` Number of cpus per task
- `-o` Name of the file that will receive the output messages.
- `-e` Name of the file that will receive the error messages.
- `--mail-user` Email that will receive the status messages of the job
- `--mail-type` What type of message Slurm should send to the email. ALL tells Slurm to send messages of events (job started, job completed, job failed, etc.)

To see all list of Slurm parameters, check: https://slurm.schedmd.com/sbatch.html.

While you can execute this command as described above, creating a new configuration file to store the values of Slurm parameters will add flexibility to your workflow. We will do it in the next

section.

Creating a cluster configuration file

Let’s create a new file called 'cluster.yaml' and also add it to Git version control.

```
touch cluster.yaml
git add cluster.yaml
```

Most of the time, Slurm jobs will all contain the same basic information, such as the name of the qos, email, amount of memory, number of cores, etc. Therefore, we can create some default parameters that will be used in all our Slurm jobs.

Copy the code below to your 'cluster.yaml' file. **Remember to change the qos to the one that you have access. Also, change the email that will receive all Slurm messages**.

```
__default__:
    cluster: slurm
    ntasks: 1
    n_of_threads: 1
    mem: 4gb
    qos: kirst
    output: logs/snakemake_%j.out
    error: logs/snakemake_%j.err
    time: 02:00:00
    email_user: wendellpereira@ufl.edu
    email_type: ALL
```

From now one, every time we submit a Slurm job through Snakemake and using this configuration file, it will use these parameters, unless specified otherwise. If we need to change some of the parameters for only one rule, we can do it by adding the parameters that you want to change, as demonstrated below:

```
mapping:
    n_of_threads: 6
    output: logs/mapping_%j.out
    error: logs/mapping_%j.err
    mem: 10gb
```

Now, if we call snakemake to execute the mapping rule, it will submit a job with all default parameters, except the ones specified to the mapping rule in the "cluster.yaml" file. **No change is necessary in the "Snakefile" or "config.yaml" files**.

Let’s commit our modifications in the "cluster.yaml" file. Then, we will learn how to execute Snakemake jobs using a configuration file for Slurm.

```
git commit "Adds the parameters for the mapping rule" cluster.yaml
```

Executing jobs on Slurm using Snakemake

As shown below, the main difference in the code is the addition of the option `--cluster-config`. It receives the name of the configuration file that contains the Slurm options. As a consequence, a new variable called `cluster` will be created, and we can used it in a similar way to what we did to the variable `params`. See below how `{cluster.qos}` is used to pass the `qos` name stored in the "cluster.yaml" file.

```
snakemake -np -j 10 --cluster-config cluster.yaml --cluster "sbatch -A {cluster.qos}
--mem={cluster.mem} -t {cluster.time} -c {cluster.n_of_threads} -e {cluster.error} -o
{cluster.output} --mail-user {cluster.email_user} --mail-type {cluster.email_type}"
mapping
```

--cluster-config specifies the location of a file containing the cluster configuration for each rule in the Snakefile.

Remove the option `n` and execute the above command. Now, it will be submit as a Slurm job.

Since all output files were already created, Snakemake will tell us that there is nothing to be done for this rule. We can add the option `--force` to force the execution of the rule. Also, deleting one or more of the outputs will make Snakemake run the rule, since it will detect that there are missing files that this rule should create.

```
snakemake --force -p -j 10 --cluster-config cluster.yaml --cluster "sbatch -A
{cluster.qos} --mem={cluster.mem} -t {cluster.time} -c {cluster.n_of_threads} -e
{cluster.error} -o {cluster.output} --mail-user {cluster.email_user} --mail-type
{cluster.email_type}" mapping
```

If you execute the command as show above, your terminal will be occupied until the job is finished. If you don't want to wait, it is possible to use `nohup` and `&` to execute the command and keep you terminal free: **nohup** *snakemake --force -p -j 10 --cluster-config cluster.yaml --cluster "sbatch -A {cluster.qos} --mem={cluster.mem} -t {cluster.time} -c {cluster.n_of_threads} -e {cluster.error} -o {cluster.output} --mail-user {cluster.email_user} --mail-type {cluster.email_type}" mapping* **&**.

Adding other rules to the pipeline and executing some interesting options of Snakemake.

Since we already went over the main concepts, we are going to skip the interpretation of the next steps of the pipeline. Using the Pertea et al, 2016 paper and the Snakemake manual, you can understand what the next rules will do. Take it as a practice exercise and try to understand them after the workshop. Maybe, you can even create your rules to finish the results described in the paper.

For now, just copy and paste the following lines in the "Snakefile".

```
rule sorting_sam:
    input:
        expand("mapping_out/{samples}.sam", samples=config["mapping"]["inputs"])
    params:
        threads=config["sorting_sam"]["params"]["threads"]
    output:
        expand("mapping_out/{samples}.bam", samples=config["mapping"]["inputs"])
    shell:
        """
        for sample in {input};
        do

            name=$(basename $sample \".sam\")

            samtools sort -@ {params.threads} -o mapping_out/${{name}}.bam $sample

        done
        """

rule assembly:
    input:
        gtf=expand("{gtf}", gtf=config["assembly"]["params"]["annotation"]),
        bam=expand("mapping_out/{samples}.bam", samples=config["mapping"]["inputs"])
    params:
        threads=config["assembly"]["params"]["threads"],
    output:
        expand("gtfs/{samples}.gtf", samples=config["mapping"]["inputs"])
    shell:
        """
        for sample in {input.bam}
        do

            prefix_name=$(basename $sample \"_chrX.bam\")

            stringtie $sample -G {input.gtf} -p {params.threads} -o
gtfs/${{prefix_name}}_chrX.gtf -l ${{prefix_name}}

        done
        """
```

The following lines in the "config.yaml" file.

```
sorting_sam:
    params:
        threads: 4

assembly:
    params:
        threads: 8
        annotation: "chrX_data/genes/chrX.gtf"
```

And the following lines in the "cluster.yaml" file.

```
    sorting_sam:
        n_of_threads: 4
        output: sorting_sam_%j.out
        error: sorting_sam_%j.err
        mem: 10gb

    assembly:
        n_of_threads: 8
        output: assembly_%j.out
        error: assembly_%j.err
        mem: 10gb
```

Don’t forget to commit the changes and synchronizing the files.

```
git commit -m "Adds new rules to create the transcripts assembly" Snakefile
git commit -m "Adds new rules to create the transcripts assembly" config.yaml
git commit -m "Adds new rules to create the transcripts assembly" cluster.yaml

git push origin master
```

Adding a rule to execute the whole pipeline.

As mentioned before, when the name of the rule is not given, Snakemake will try to execute the first rule in the "Snakefile". Therefore, an easy way to execute your pipeline is to create a rule that anticipates all files generated from your output. Or, a rule that anticipates a file which depends of the execution of all rules.

In our case, lets use the output of the last rule as the input. We usually call this rule "all", and it has the following structure (Note that no output is necessary for this rule).

```
rule all:
    input:
        #mapping
        expand("mapping_out/{samples}.sam", samples=config["mapping"]["inputs"]),
        #sorting_sam
        expand("mapping_out/{samples}.bam", samples=config["mapping"]["inputs"]),
        #assembly
        expand("gtfs/{samples}.gtf", samples=config["mapping"]["inputs"])
```

Copy the rule above in the top of your Snakefile, right after the command that loads the modules.

Now, let's execute a dry-run of this rule. Since we already have these outputs, nothing will be done. So, to visualize the list of rules that will be executed by `all` rule, we can add the option `--forceall`.

Execute the below command.

```
snakemake -np --forceall all
```

or

```
snakemake -np --forceall
```

Using Snakemake, it is possible to create some reports and summaries of your workflow. They can be very useful for keep tracking of what each rule is generating, execution time and other metrics. Let's generate a report of our workflow.

```
snakemake --report uf_course.html all
```

Now, let's test how to use the `--summary` option.

```
snakemake --summary
snakemake --summary --forcerun assembly
```

Final remarks

Snakemake is still under very active development, with new options been released frequently. Keep updating your knowledge by visiting the manual at: https://snakemake.readthedocs.io. There are also many other sources on internet where you can learn how to apply Snakemake in more advanced ways.

I hope this workshop and tutorial was useful to give you good ideas for implementing Snakemake in your routinely activities. Below are some links for more advanced features that can be useful in your workflow.

- Temporary and protected files
- Modularization
- Using –cluster-status
- Dealing with very large workflows
- Remote files