TRAINER'S MANUAL

Implementing Scalable Bioinformatic Workflows in Snakemake and Nextflow

Nathan S. Watson-Haigh
Radosław Suchecki

Across Australia

Aug/Sep 2019

TRAINER'S MANUAL

Licensing

Contents

Workshop Information

The Trainers

Dr. Nathan S. Watson-Haigh
Senior Bioinformatician
Bioinformatics Hub, University of Adelaide
nathan.watson-haigh@adelaide.edu.au

Dr. Radosław Suchecki
Research Scientist
Crop Bioinformatics and Data Science, CSIRO
rad.suchecki@csiro.au

Providing Feedback

While we endeavour to deliver a workshop with quality content and documentation in a venue conducive to an exciting, well run hands-on workshop with a bunch of knowledgeable and likable trainers, we know there are things we could do better.

Whilst we want to know what didn't quite hit the mark for you, what would be most helpful and least depressing, would be for you to provide ways to improve the workshop. i.e. constructive feedback. After all, if we knew something wasn't going to work, we wouldn't have done it or put it into the workshop in the first place!

Clearly, we also want to know what we did well! This gives us that "feel good" factor which will see us through those long days and nights in the lead up to such hands-on workshops!

With that in mind, we'll provide a some high tech mechanism through which you can provide anonymous feedback during the workshop:

1. Some empty ruled pages at the back of this handout. Use them for your own personal notes or for writing specific comments/feedback about the workshop as it progresses.

Document Structure

We have provided you with an electronic copy of the workshop's hands-on tutorial documents. We have done this for two reasons: 1) you will have something to take away with you at the end of the workshop, and 2) you can save time (mis)typing commands on the command line by using copy-and-paste.

We advise you to use Acrobat Reader to view the PDF. This is because it properly supports some features we have implemented to ensure that copy-and-paste of commands works as expected. This includes the appropriate copy-and-paste of special characters like tilde and hyphens as well as skipping line numbers for easy copy-and-past of whole code blocks.

While you could fly through the hands-on sessions doing copy-and-paste, you will learn more if you use the time saved from not having to type all those commands, to understand what each command is doing!

The commands to enter at a terminal look something like this:

```
1 tophat --solexa-quals -g 2 --library-type fr-unstranded -j \
    annotation/Danio_rerio.Zv9.66.spliceSites -o tophat/ZV9_2cells \
    genome/ZV9 data/2cells_1.fastq data/2cells_2.fastq
```

The following styled code is not to be entered at a terminal, it is simply to show you the syntax of the command. You must use your own judgement to substitute in the correct arguments, options, filenames etc

```
tophat [options]* <index_base> <reads_1> <reads_2>
```

The following is an example of how R commands are styled:

```
R --no-save
library(plotrix)
data <- read.table("run_25/stats.txt", header=TRUE)
weighted.hist(data$short1_cov+data$short2_cov, data$lgth, breaks=0:70)
q()
```

The following icons are used in the margin, throughout the documentation to help you navigate around the document more easily:


Important


For reference


Follow these steps


Questions to answer


Warning - STOP and read


Bonus exercise for fast learners


Advanced exercise for super-fast learners

Introduction to Snakemake

Primary Author(s):
Nathan S. Watson-Haigh nathan.watson-haigh@adelaide.edu.au

Contributor(s):

Key Learning Outcomes

After completing this module the trainee should be able to:

- Install Snakemake in a conda environment
- Execute a Snakemake workflow
- Use the provided "profile" to execute jobs on a compute cluster
- Write simple Snakemake rules capable of generating some output(s) by executing some code which perates on some input(s)

Resources Required

For the purpose of this training you need access to:

- A compute cluster with the `module` command available to you for loading software
- Singularity (https://sylabs.io/singularity/) - available as a module on the above cluster
- Conda(https://www.anaconda.com/distribution/) - available as a module on the above cluster

Tools Used

Snakemake
https://snakemake.readthedocs.io

Graphviz
https://www.graphviz.org

Useful Links

Slurm Documentation
https://slurm.schedmd.com/documentation.html

Setting Up Your Environment

For the purpose of the workshop we will be working on the head node of an HPC cluster running slurm (https://slurm.schedmd.com/documentation.html). This is the most likely infrastructure that fellow bioinformaticians already find themselves using on a regular basis. We also assume that the cluster provides the `module` command for you to load software and the modules `Anaconda3` and `Singularity` are available to use.

The execution of the Snakemake workflow will actually take place on the cluster head node with jobs being submitted to Slurm for queing and processing. From the head node, Snakemake will monitor the submitted jobs for their completion status and submit new jobs as dependent jobs complete sucessfully.

Connect to the Cluster Head Node

First up, lets connect to the head node of the HPC cluster using `ssh`.

See your local facilitator for connection details. You will have one user account per person.

Monitoring Slurm Jobs

You can monitor all jobs in the slurm squeue, or just your own job(s) using the slurm command `squeue`:

```
# All jobs in the queue
squeue

# Just your own jobs
squeue --user ${USER}
```

For convienience we have provided you with the `sq` function which produces nicer output than the default `squeue` and only shows your own jobs:

```
# Your own jobs
sq

# Someone elses jobs
sq --user ${SOMEONE_ELSE}
```

Install Snakemake

The recommended installation route for Snakemake is through a conda environemnt (https://snakemake.readthedocs.io/en/stable/getting_started/installation.html). As such, you need Anaconda3, usually avaiable to you on your cluster via the module system.

```
# We use a specific version for reproducibility reasons
# Find the latest version: https://anaconda.org/search?q=snakemake
SNAKEMAKE_VERSION="5.5.4"

# Load miniconda
module load \
  miniconda3-4.6.14-gcc-5.4.0-kkzv7zk

#####
# One-time commands
#####
# Integrate conda into bash
conda init bash
. ~/.bashrc

# Change the default location into which conda saves packages
# and environments
conda config --prepend pkgs_dirs /shared/${USER}/.conda/pkgs
conda config --prepend envs_dirs /shared/${USER}/.conda/envs

# Change the default channels used for finding software and
# resolving dependencies
conda config --add channels defaults
conda config --add channels bioconda
conda config --add channels conda-forge
#####
```

Do NOT run the following command! This is provided for future reference so you know how to Install Snakemake on another system. Rather than creating the conda environment from scratch, we'll simply copy a pre-existing directory so we save time, and possible headaches.

```
# Install snakemake using conda
# This might take 5-10mins
conda create \
  --name snakemake \
  --yes \
  snakemake=${SNAKEMAKE_VERSION:-5.5.4}
```

Snakemake installation is now complete.

For the purposes of this workshop, simply copy the following `.conda` directory and you will have Snakemake setup and ready to go:

```
mkdir --parents /shared/${USER}
cp --recursive \
  /shared/ubuntu/.conda \
  /shared/${USER}/
```

All that is left to do is to activate the environment which will make `snakemake` available on the command line:

```
# Activate the newly created conda environment
conda activate snakemake
```

Integrate Snakemake autocompletion into bash:

```
complete -o bashdefault -C snakemake-bash-completion snakemake
```

Test if Snakemake is actually working:

```
snakemake --version
```

If you experience problems with the installation, head to the Troubleshooting section for help.

While waiting for others to catch up, why not have a look into how you would go about updating `Snakemake` within this conda environment if there is a new version available.

```
conda update \
  snakemake
```

Your First Minimal Snakefile

To get started with Snakemake, all you need to do is create a `Snakefile` (note the capitalisation) containing a rule which specifies how to create an output file.

Setup a working directory for this task:

```
mkdir --parents /shared/${USER}/snakemake/minimal
cd /shared/${USER}/snakemake/minimal
```

Create a file called `Snakefile` and add the following content:

```
rule hello_world:
  output:
```

```
    "Hello/World.txt",
  shell:
    """
    echo "Hello, World!" > {output}
    """
```

You can now run this workflow in one of 3 ways:

```
# Request Snakemake to generate the specific output file
# "Hello/World.txt"
snakemake Hello/World.txt

# Request Snakemake to execute the specific rule "hello_world"
snakemake hello_world

# Request Snakemake to execute the first rule in the Snakefile
snakemake
```

What happens if you run one of the above commands two or more times? Why?

Snakemake found the requested output file was already there so decided not to regenerate it.

Generalising Rules with Wildcards

The original `hello_world` rule wasn't very flexible. We couldn't say "Hello, World!" in Spanish, Polish or French. However, we can generalise the rule using "wildcards":

```
rule hello_world:
  output:
    "{cheer}/{world}.txt",
  shell:
    """
    echo "{wildcards.cheer}, {wildcards.world}!" > {output}
    """
```

Now we can use whatever language we want:

```
# In English
snakemake Hello/World.txt

# In Polish
snakemake Czesc/Swiat.txt

# In French and Spanish at the same time
```

```
snakemake Monde/Monde.txt Ciao/Mondo.txt
```

Take a look at the files created:

```
tree ./
```

Submitting Jobs to Slurm

By default, Snakemake executes jobs on the same computer on which it is running. For Snakemake to be able to submit jobs to a cluster resource management/queuing system, such as Slurm, we can use a "profile" which convieniently contains scripts for job submission and monitoring as well as setting some additional Snakemake command line arguments so it can "talk" to a cluster backend.

To avoid having to delve into implementing our own "profile" for use with our Slurm cluster, we have created a Slurm profile ready for you to use. So lets grab it:

```
# Ensure a working directory exists and move into it
mkdir --parents /shared/${USER}/snakemake/tutorial
cd /shared/${USER}/snakemake/tutorial

# Clone the Snakemake template repository from GitHub
git clone https://github.com/UofABioinformaticsHub/snakemake_template ./

# Checkout the "simple" branch
git checkout simple
```

The `Snakefile` in this branch of the repository is the same "Hello, World!" example you created above, with wildcards. Lets see how we use the provided "profile" to get Snakemake to submit jobs to Slurm:

```
snakemake \
  --profile profiles/slurm \
  Hello/World.txt Czesc/Swiat.txt Monde/Monde.txt Ciao/Mondo.txt
```

If the `STDOUT` and `STDERR` of the command(s) in a rule are not explicitly sent to a file, then they will end up in Slurm's log file for a particular job which is normally something like `slurm-<job_id>.out`. This isn't that helpful for debugging purposes, so the provided profile changes this to `logs/<rule_name>/<wildcards>.out` e.g. `logs/hello_world/cheer=Ciao,world=Mondo.out`.

Cleanup after yourself!

```
snakemake \
  --delete-all-output \
  Hello/World.txt Czesc/Swiat.txt Monde/Monde.txt Ciao/Mondo.txt
```

A Bash "Pipeline"

A bioinformatic "pipeline" is commonly a single, monolithic bash script which performs all the tasks which need to be performed. For example, someone might have written a script for performing the following tasks:

- Run FastQC across all the raw read files
- Adapter, quality, and read length filtering using Trimmomatic
- Aggregating FastQC reports from the raw reads using MultiQC
- Index the reference FASTA file
- Perform a `bwa-mem` read alignment

We have this script available for you on the `tutorial` branch, switch to it and have a look:

```
1 git checkout tutorial
2
3 less analysis.sh
```

While the author of such a script should be commended for their efforts in documenting their analysis using a script, it has several significant limitations:

Not parallelised
loops over input files, executing independant commands in sequential order

Resources over-specified
the compute resources needed by the script are dictated by the command(s) with the largest requirement(s)

Not idempotent
significant programming logic is needed to wrap around commands to detect failures and only execute parts of the analysis which failed in earlier attempts

How might ***you*** modify the above script to:

- Add new samples
- Rerun the script if you find one of the files generated is corrupt
- Include readgroup information at the `bwa-mem` step (`-b` argumanet)

How would you avoid rerunning commands which take a long time and already completed sucessfully on a previous run e.g. the reference index, `bwa-mem` etc?

With difficulty! Enter - workflow management systems like Snakemake or Nextflow.

Reimplementing A Workflow in Snakemake

We will walk you through the steps of reimplementing the first few steps of the above script into a Snakemake workflow. Along the way, we will introduce the core concepts of Snakemake and then ask you to reimplement the `bwa-mem` step yourselves. For those working quickly, you will have the opportunity to reimplement the `multiqc` step. This will provide you with a foundation for you to be able to convert your own workflows into Snakemake rules and begin reaping the rewards of being able to run your analyses in Snakemake.

Getting the Data

We've provided you with some real data whole genome sequencing (WGS) data from wheat together with a small chunk of the wheat genome. The data set is small enough so each step in the analysis will take less than a couple of minutes to run. We have a copy of this data available locally to save on badnwidth, time and the possibility we are detected as a DDoS attack on some poor remote server!

```
# Get a copy of the data
cp --recursive \
  /shared/data/{raw_reads,references,misc} \
  ./

# Have a look at what files we'd provided
tree raw_reads references misc
```

Implementing BWA Indexing

We have provided an out-of-the-box `Snakefile` capable of indexing the provided reference sequence, together with comments. Lets have a bit of a play before we get around to actually running the workflow.

```
less Snakefile

# All these commands have the same effect
snakemake --dryrun bwa_index
snakemake --dryrun all
snakemake --dryrun
```

The effect `--dryrun` is to simply show you what "would" be run, without actually running it. It's useful to ensure you're going to get what you though, especially as your worflows get larger and more interconnected.

Another useful feature is to generate a directed acyclic graph (DAG) of the jobs which comprise the workflow and how they are linked together. Although for this workflow is not yet that impressive, but we'll have a look at how we generate the DAG:

```
snakemake \
  --dag \
| dot -Tpdf \
> dag1.pdf
```

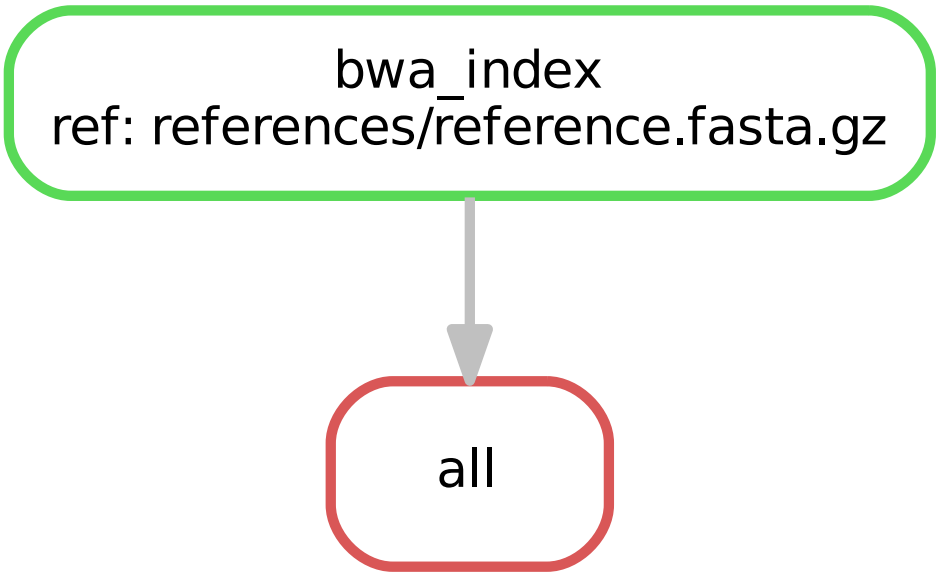

Figure 1: DAG of jobs showing `bwa_index` job dependant on the `all` pseudo-rule.

Implementing FastQC

Lets add a rule for performing FastQC on our input files. Looking at the `analysis.sh` file we see the following command is executed for each `SAMPLE` while interating over the `SAMPLES` array:

```
fastqc --threads 1 \
  raw_reads/${SAMPLE}_R1.fastq.gz \
  raw_reads/${SAMPLE}_R2.fastq.gz
```

This command can be converted into a Snakemake `rule` by adding the following rule to the `Snakefile`:

```
rule fastqc:
  input:
    r1 = "raw_reads/{SAMPLE}_R1.fastq.gz",
    r2 = "raw_reads/{SAMPLE}_R2.fastq.gz",
  output:
    zip = [ "raw_reads/{SAMPLE}_R1_fastqc.zip", \
        "raw_reads/{SAMPLE}_R2_fastqc.zip" ],
    html = [ "raw_reads/{SAMPLE}_R1_fastqc.html", \
        "raw_reads/{SAMPLE}_R2_fastqc.html" ],
  shell:
    """
    fastqc --threads 1 {input.r1} {input.r2}
    """
```

Now we can run Snakemake and request a "target" file which matches an output files defined by the above `fastq` rule:

```
snakemake --dryrun raw_reads/ACBarrie_R1_fastqc.html \
    raw_reads/ACBarrie_R2_fastqc.html
```

There are a few improvements we can make to this rule:

- We don't need to process both the R1 and R2 read files with the same FastQC job. We can operate on one read file at a time. By doing this, Snakemake will be able to execute the FastQC job for each file in parallel.
- We want a convienient way of generating FastQC outputs for ALL samples without typing them all at the command line.

Improving FastQC Parallelisation

Add the following new rule to your `Snakefile`:

```
rule fastqc_single_input:
  input:
    "raw_reads/{prefix}.fastq.gz",
  output:
    zip = "raw_reads/{prefix}_fastqc.zip",
    html = "raw_reads/{prefix}_fastqc.html",
  shell:
    """
    fastqc --threads 1 {input}
    """
```

Now run the same Snakemake dryrun command as before:

```
snakemake --dryrun raw_reads/ACBarrie_R1_fastqc.html \
    raw_reads/ACBarrie_R2_fastqc.html
```

Why did Snakemake complain about an `AmbiguousRuleException`?

We now have two rules (`fastqc` and `fastqc_single_input`) capable of generating the two files we requested and it doesn't know which it should use.

How could we fix it? Hint: https://snakemake.readthedocs.io/en/stable/snakefiles/rules.html#handling-ambiguous-rules

Three options:

- Delete the old `fastqc` as the `fastqc_single_input` is superior
- Use `ruleorder` to define precedence
- Use `--allow-ambiguity` so Snakemake uses the first rule encountered

Go ahead and delete the `fastqc` rule in favour of the `fastqc_single_input` rule. Do the same dryrun and see how many jobs Snakemake would run in order to create those files:

```
snakemake --dryrun raw_reads/ACBarrie_R1_fastqc.html \
    raw_reads/ACBarrie_R2_fastqc.html
```

Pseudo-Rules

We can use "pseudo-rules" to define a list of target filenames for creation when we use the rule name as a "target". Pseudo-rules consist of just an `input` directive:

```
rule all:
  input:
    "raw_reads/ACBarrie_R1_fastqc.html",
    "raw_reads/ACBarrie_R2_fastqc.html",
```

By convention, the first pseudo-rule in the `Snakefile` is called `all` and specifies all the output filenames of the workflow. This now means we can execute a workflow in any of the following ways:

```
# Not specifying a target will result in Snakemake executing the
# first rule in the Snakefile ("all" in this case)
snakemake --dryrun

# Explicityly request the "all" rule
snakemake --dryrun all
```

When workflows get larger and the lists of filenames get bigger, specifying long lists of filenames in pseudo-rules can start to feel cumbersome. Since Snakemake syntax is an extension of Python, we can start to use some Python data structures and functions to help.

Add the following Python array of sample names (with most commented out for now) at the top of the file:

```
SAMPLES = [
  "ACBarrie",
  "Alsen",
# "Baxter",
# "Chara",
# "Drysdale",
# "Excalibur",
# "Gladius",
# "H45",
# "Kukri",
# "Pastor",
# "RAC875",
# "Volcanii",
# "Westonia",
# "Wyalkatchem",
# "Xiaoyan",
# "Yitpi",
]
```

Add FastQC output files for all samples in the `SAMPLES` array, as well as both read files, as

new targets to the existing `all` rule. We'll make use of the `expand()` function to simplify things somewhat. The resulting `all` pseudo-rule should look like this:

```
rule all:
  input:
    expand("references/reference.fasta.gz.{ext}", ext=['amb', 'ann', \
        'bwt', 'pac', 'sa']),
    expand("raw_reads/{SAMPLE}_{read}_fastqc.html", SAMPLE=SAMPLES, \
        read=['R1', 'R2']),
```

Lets take a look at what jobs would be run if we run the whole workflow. Remember, the following commands are equivilent:

```
# Explicitly run the "all" pseudo-rule
snakemake --dryrun all

# Run the first rule in the Snakefile. This should be the "all" rule by \
    convention
snakemake --dryrun
```

Lets look at the DAG for the workflow:

```
snakemake \
  --dag \
| dot -Tpdf \
> dag2.pdf
```

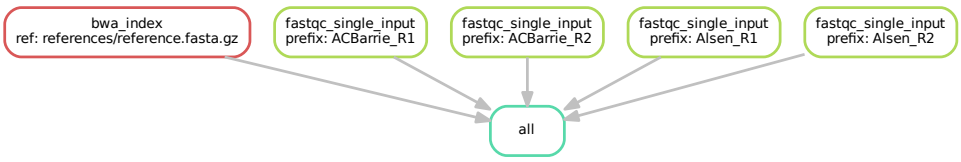

Figure 2: DAG of jobs showing `bwa_index` and several `fastqc_single_input` job dependant on the `all` pseudo-rule.

Executing the Workflow on Slurm

Up until now, we've just been playing around with `--dryrun`, so lets move on and start executing the workflow on the Slurm cluster!

Remember, we need to use the Slurm profile we've provided you with so Snakemake knows how to communicate with Slurm. In addition, we're also going to execute the jobs within a singularity container which has the tools we need already installed inside it.

```
# Make sure Singularity is available
module load \
  singularity-3.2.1-gcc-5.4.0-tn5ndnb

```

```
# Execute the workflow
snakemake \
  --profile profiles/slurm \
  --use-singularity
```

If you're quick you might see the progress of your jobs in the Slurm queue but executing this in another window:

```
sq
```

Implementing Trimmomatic

We've gone through implementing the FastQC command as a Snakemake rule and demonstrated the core concepts of Snakemake along the way. We'll go through implementing one more command as a Snakemake rule before you go off and try one on your own!

```
rule trimmomatic:
  input:
    r1 = "raw_reads/{SAMPLE}_R1.fastq.gz",
    r2 = "raw_reads/{SAMPLE}_R2.fastq.gz",
    adapters = "misc/trimmomatic_adapters/TruSeq3-PE.fa"
  output:
    r1 = "qc_reads/{SAMPLE}_R1.fastq.gz",
    r2 = "qc_reads/{SAMPLE}_R2.fastq.gz",
    r1_unpaired = "qc_reads/{SAMPLE}_R1.unpaired.fastq.gz",
    r2_unpaired = "qc_reads/{SAMPLE}_R2.unpaired.fastq.gz",
  shell:
    """
    trimmomatic PE \
      -threads 1 \
      {input.r1} {input.r2} \
      {output.r1} {output.r1_unpaired} \
      {output.r2} {output.r2_unpaired} \
      ILLUMINACLIP:{input.adapters}:2:30:10:3:true \
      LEADING:2 \
      TRAILING:2 \
      SLIDINGWINDOW:4:15 \
      MINLEN:36
    """
```

Next, we need to add the trimmomatic output files to our `all` pseudo-rule to make it convenient to create them. Your `all` pseudo-rule should look like this:

```
rule all:
  input:
    expand("references/reference.fasta.gz.{ext}", ext=['amb', 'ann', \
        'bwt', 'pac', 'sa']),
```

```
    expand("raw_reads/{SAMPLE}_{read}_fastqc.html", SAMPLE=SAMPLES, \
       read=['R1', 'R2']),
    expand("qc_reads/{SAMPLE}_{read}.fastq.gz", SAMPLE=SAMPLES, \
       read=['R1', 'R2']),
```

Legacy Stuff From Here Onwards

```
# Ensure a working directory exists and move into it
mkdir -p /shared/${USER}/snakemake-tutorial
cd /shared/${USER}/snakemake-tutorial

# Clone the Snakemake template repository from GitHub
git clone https://github.com/UofABioinformaticsHub/snakemake_template ./
```

Add the raw read data and reference files:

```
cp --recursive --symbolic-link /shared/data/* ./
```

Setup a conda software environment for the analysis

```
# Load miniconda
module load \
  miniconda3-4.6.14-gcc-5.4.0-kkzv7zk

# Install snakemake by submitting a job to slurm
# This will take ~5mins
sbatch --job-name 'tool-install' --mem 4G --time 15:00 --wrap \
 "conda create \
  --name tutorial \
  --channel bioconda --channel conda-forge \
  --yes \
  fastqc=0.11.8 \
  multiqc=1.7 \
  trimmomatic=0.36 \
  pigz=2.3.4 \
  bwa=0.7.17 \
  samtools=1.9 \
  htslib=1.9"
```

Submit the script:

```
#sbatch analysis.sh
```

Your First Snakefile

```
rule fastqc_a_file:
```

```
2   input:
3     "data/0Hour_001_1.fq.gz",
4   output:
5     "data/0Hour_001_1_fastqc.html",
6     "data/0Hour_001_1_fastqc.zip",
7   conda:
8     "envs/default"
9   shell:
10    """
11    fastqc data/0Hour_001_1.fq.gz
12    """
```

Snakemake basics

The way Snakemake runs can be thought of as being somewhat like a satellite navigation system, you request a destination ("target") and the system works out a route to get you there using its knowledge of the roads ("dependancies") which link together towns ("files").

To exemplify the basics of Snakemake, we're going to take a look at constructing a "Hello, world!" example. We will go through the process of explicitly defining "rules", the basic building blocks of a Snakemake workflow. Rules are used to define how to create output ("target") files, possibly requiring some `input` files. We will then work to generalise these rule, introducing other Snakemake concepts and features along the way.

Rule Definition

By convention, we use a file called `Snakefile` in which to define our workflow rules. "Rules" define how to create output (target) files. For example:

```
1 rule hello_world:
2   output:
3     "Hello/World.txt",
4   shell:
5     """
6     echo "Hello, World!" > Hello/World.txt
7     """
```

This rule has the name `hello_world` (line 1). The `output` file (line 3) is created by a `shell` script (line 6) which is simply echoing a string, `Hello, world!`, and redirecting that into the appropriately named output file.

Create a working directory under `/shared/${USER}`, and add the above `hello_world` rule to a file called `Snakefile`:

```
1 # Create a working directory and move into it
```

```
mkdir -p /shared/${USER}/my_hello_world
cd /shared/${USER}/my_hello_world

# Create the Snakefile with the relevant contents using this heredoc
cat >Snakefile <<EOF
rule hello_world:
  output:
    "Hello/World.txt",
  shell:
    """
    echo "Hello, World!" > Hello/World.txt
    """
EOF
```

Things to be careful of:

- Correct capitalisation of the filename `Snakefile` - this saves you from having to do something like `snakemake --snakefile my_oddly_named_Snakefile`
- Consistent indentation (tabs or spaces but not both) of the Snakemake directives at the first indentation level. This is because a Snakemake syntax is an extension of Python which has this constraint.

Now execute the workflow

```
snakemake
```

Try executing it for a second time:

```
snakemake
```

A few interesting thing to note at this point is that Snakemake will:

- Use the `Snakefile` as it's input unless you use `--snakefile my_snakefile`
- Execute the first rule encountered in the `Snakefile`
- Create the required output directory structure, in this case creating the `Hello/` directory
- Will not execute rules for which the `target` output file already exists unless `--force`, `--forceall` or `--forcerun` is used.

`Snakemake` executes the first rule in the `Snakefile` unless we explicityly ask for a "target". This means the following are equivilent:

```
# Not specifying a target will result in Snakemake executing the
# first rule in the Snakefile ("hello_world" in this case)
snakemake

# Explicitly requesting the "hello_world" rule to be run:
snakemake hello_world

# Explicityly request the target filename to be created:
snakemake Hello/World.txt
```

Generalising Rules

The above `hello_world` rule contains some redundancy, namely the output filename (`Hello/World.txt`) is specified in two places. This would mean we would have to make modifications in two places if we wanted to change the name of the output file. Snakemake allows us to refer to the values of other elements of the rule using a curley brace syntax within the `shell` directive.

So, rewrite the rule as follows:

```
rule hello_world:
  output:
    "Hello/World.txt",
  shell:
    """
    echo "Hello, World!" > {output}
    """
```

{`output`} used within the `shell` directive will be substituted for the filename defined in the `output` directive. Before running the workflow again, we will first request Snakemake to delete all the output files associated with a target. The target could be either implied or explicitly provided. The following are equivilent:

```
# Delete output files of the implied target (the first rule in the \
    Snakefile)
snakemake --delete-all-output

# Delete output files of the explicitly defined "hello_world" rule:
snakemake --delete-all-output hello_world

# Delete output files of the explicityly requested target filename:
snakemake --delete-all-output Hello/World.txt
```

What if we wanted to write a “Hello, world!” rule capable of saing “Hello” in different lannguages? We could include a new rule:

```
rule hello_world:
  output:
```

```
    "Hello/World.txt",
  shell:
    """
    echo "Hello, World!" > {output}
    """

rule hello_world_spannish:
  output:
    "Hola/World.txt",
  shell:
    """
    echo "Hola, World!" > {output}
    """
```

This isn't good as we have introduced a whole lot of redundancy by mostly doing a copy and paste and then changing a couple of words! There is a better way.

Snakemake uses named "wildcards" to capture strings defined in the `output` filenames and a mechanism to refer back to them elsewhere within a rule. See here:

```
rule hello_world:
  output:
    "{cheer}/World.txt",
  shell:
    """
    echo "{wildcards.cheer}, World!" > {output}
    """
```

We define a named wildcard called `cheer`, using curley brace syntax within the filename of the `output` directive. We can then reuse the value of this wildcard within the `shell` directive via {`wildcards.cheer`}.

Modify your `Snakefile` to contain the above rule and try executing the workflow using each of the following:

```
# Not specifying a target will result in Snakemake executing the
# first rule in the Snakefile ("hello_world" in this case)
snakemake

# Explicitly requesting the "hello_world" rule to be run:
snakemake hello_world
```

Why did these two commands not work this time around?

Essentially, `cheer` is undefined. Because of this generalisation, we must not explicitly provide the target filenames.

How would you now get Snakemake to create the `Hello/World.txt` target file?

```
# Explicityly request the target filename to be created:
snakemake Hello/World.txt
```

How would you now get Snakemake to create the `Hola/World.txt` target file?

```
# Explicityly request the target filename to be created:
snakemake Hola/World.txt
```

Pseudo-Rules

We have now see how wildcards can be used to generalise rules. This is a powerful feature of Snakemake, but this means we lose the convienience of being able to specify a rule name as the target since the wildcards are essentially undefined in that situation. We have also seen that if we do not explicitly specify a target, then Snakemake attempts to execute the first rule in the `Snakefile`. Unfortunately, if that rule makes use of wildcards, and error is thrown.

We can use pseudo-rules to define a list of target filenames for creation. These filenames are specified using the `input` directive only:

```
rule all:
  input:
    "Bonjour/World.txt",
    "Ciao/World.txt",
    "Hello/World.txt",
    "Hola/World.txt",

rule hello_world:
  output:
    "{cheer}/World.txt",
  shell:
    """
    echo "{wildcards.cheer}, World!" > {output}
    """
```

By convention, the first pseudo-rule in the `Snakefile` is called `all` and specifies all the output filenames of the workflow. This now means we can execute this workflow in any of the following ways:

```
# Not specifying a target will result in Snakemake executing the
# first rule in the Snakefile ("all" in this case)
```

```
snakemake

# Explicityly request the "all" rule
snakemake all

# Explicitly a cheer in Polish
snakemake czesc/World.txt
```

When workflows get larger and the lists of filenames get bigger, specifying long lists of filenames in pseudo-rules can start to feel cumbersome. Since Snakemake syntax is an extension of Python, we can start to use some Python data structures and functions to help:

```
cheers = ['Bonjour', 'Ciao', 'Hello', 'Hola']

rule all:
  input:
    expand("{cheer}/World.txt", cheer=cheers),

rule hello_world:
  output:
    "{cheer}/World.txt",
  shell:
    """
    echo "{wildcards.cheer}, World!" > {output}
    """
```

In the above example, we define a Python `list` called `cheers` which contains various translations of "Hello" in different languages. We then use the `expand` function in the `input` directive of the `all` rule to reconstitute a list of target filenames containing those translations available in the `cheers list`.

The use of multiple wildcards within the output filenames provide a very powerful approach to generalising rules. However, they can also start to complicate things when using "expand" to reconstitute a list of filenames using multiple wildcards. For instance, if we rerite our "Hello, World!" example to also translate the "World" word we might think to do it like this:

```
cheers = ['Bonjour', 'Ciao', 'Hello', 'Hola']
worlds = ['Monde', 'Mondo', 'World', 'Mundo']

rule all:
  input:
    expand("{cheer}/{world}.txt", cheer=cheers, world=worlds),

rule hello_world:
```

```
  output:
    "{cheer}/{world}.txt",
  shell:
    """
    echo "{wildcards.cheer}, {wildcards.world}!" > {output}
    """
```

Before executing the workflow, have a think about these questions:

How many times do you think the `hello_world` rule will be executed? 4 times or 16 times?

Execute the workflow using `--dryrun` to see what would be executed and if you answered the above question correctly:

```
snakemake --dryrun
```

16 times - By default, `expand` will generate all combinations of the values stored within the specified wildcards (4*4=16 combinations in this case)

We can force `expand` to only combine the first elements of the lists together, the second elements of the lists and so on. There by creating four combinations of the translated words. We do this by passing Python's `zip` function as the second positional argument to `expand`:

```
cheers = ['Bonjour', 'Ciao', 'Hello', 'Hola']
worlds = ['Monde', 'Mondo', 'World', 'Mundo']

rule all:
  input:
    expand("{cheer}/{world}.txt", zip, cheer=cheers, world=worlds),

rule hello_world:
  output:
    "{cheer}/{world}.txt",
  shell:
    """
    echo "{wildcards.cheer}, {wildcards.world}!" > {output}
    """
```

Take-Home Message

TODO

Example Workflow

So far, we have only looked at rules which did not specify any `input` files. This means the `output` files are not dependant on any other files. We're now going to start looking an an example bioinformatics workflow which builds on the concepts you have already covered. This example workflow consists of the following steps:

- Running FastQC across raw read files
- Aggregating raw read FastQC reports using MultiQC
- Performing adapter, quality, and read length filtering using Trimmomatic
- Running FastQC across the QC'd reads
- Aggregating the QC read FastQC reports using MultiQC
- Index the reference FASTA file
- Perform a `bwa-mem` read alignment

Lets set up the example workflow:

```
# Ensure a working directory exists and move into it
mkdir -p /shared/${USER}/snakemake-tutorial
cd /shared/${USER}/snakemake-tutorial

# Clone the example workflow repository from GitHub
git clone https://github.com/UofABioinformaticsHub/snakemake-tutorial ./
git checkout ronin-test
```

Typically, you will probably already have your raw data files available on your filesystem. To simulate this scenario, we have provided all the relevant files for you under `/shared/data/`. The example workflow expects the raw FASTQ files to exist under a `./raw_reads/` subdirectory and the reference genome file to be under the `./references/` subdirectory. Instead of copying the files from `/shared/data/` and wasting valuable filesystem space, it is recommended that you mirror the provided directory structure in your current working directory (`/shared/${USER}/snakemake-tutorial`) using symlinks. This can be achieved by running:

```
cp --recursive --symbolic-link /shared/data/* ./
```

Lets start by having a look at what the workflow looks like, the jobs which needs to be

executed etc.

```
snakemake \
  --dryrun
```

How many `bwa_mem`, `fastqc_trimmed` and total jobs need to be run as part of this workflow?

`bwa_mem`: 16 `fastqc_trimmed`: 32 Total: 101

Using the Snakemake help, what commandline argument can be used to get Snakemake to print the shell commands associated with each job? Once you know the answer try using it in conjunction with `--dryrun` to see how the Snakemake reporting has changed.

```
snakemake \
  --dryrun \
  --printshellcmds
```

Graphs of Jobs and Rules

While it's nice to see all the information about the jobs Snakemake will execute, sometimes a picture is worth a thousand words. So, lets look at two types of graphs which Snakemake can generate:

- Directed Acyclic Graph (DAG) of the jobs (`--dag`)
- Rule graph showing dependencies between rules (`--rulegraph`)

Snakemake generates graphs in the `dot` notation, a plain-text file format commonly used for describing the nodes and edges connecting them. The output can piped into Graphviz' `dot` program to convert create a `PDF`:

```
snakemake \
  --dag \
| dot -Tpdf \
> dag.pdf
```

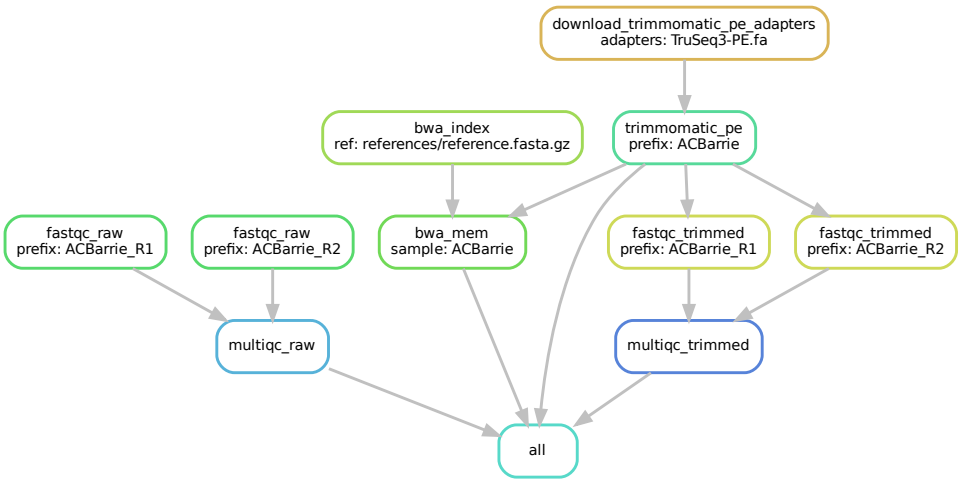

Figure 3: DAG of jobs where `accession` is `ACBarrie`.

The DAG's can get very large, very quickly. Uncomment the `Alsen` and `Baxter` lines in the `ACCESSIONS` list at the top of the `Snakefile` and generate a new DAG. It will start to look like this:

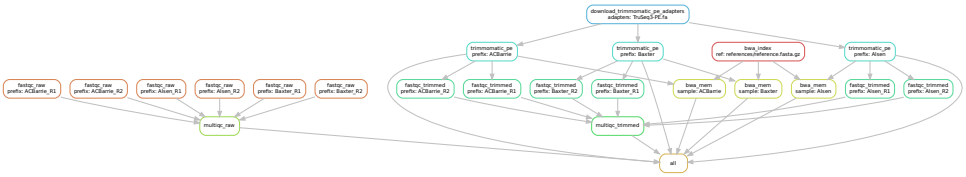

Figure 4: DAG of jobs where `accession` is `ACBarrie`, `Alsen` or `Baxter`.

Often it is enough to just look at the "rulegraph" which only contains information about the rules and the dependencies which exist between them:

```
snakemake \
  --rulegraph \
| dot -Tpdf \
> rulegraph.pdf
```

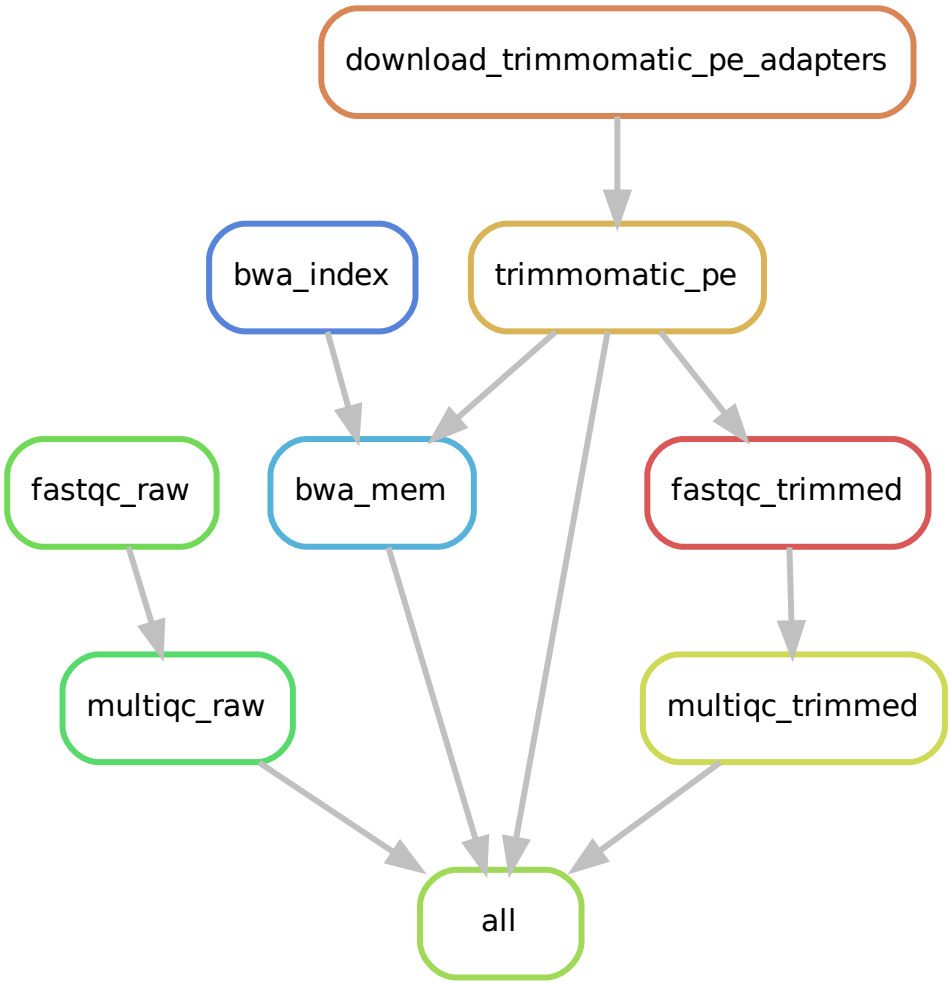

Figure 5: Graph of rules.

Executing Jobs on an HPC Cluster

So far we have been running snakemake on the cluster head node and any jobs it has been executing have also been on the head node. So far, this has been OK since the jobs have been small and short-lived. However, individual jobs of a "real-life" bioinformatics workflow will have considerably larger resource (CPU and RAM) requirements. If you execute such resource hungry jobs on the head node, your cluster sysadmin will start to growl at you!

Snakemake does not officially integrate support any one execution backend. However, it does provide convienient hooks in the form of commandline arguments for specifying scripts for interacting with an execution backend (i.e. scripts for job submission, job status and job execution). The details of this is beyond the scope of this workshop. However, we provide these in the form of a Snakemake "profile" and demonstrate how this can be used for submitting jobs to a Slurm execution backend. In addition to this profile, we also provide files for setting default cluster configuration options such as job name, resource requirements etc.

The basic command to ensure jobs are submitted to Slurm for execution are as follows:

```
snakemake \
  --profile profiles/slurm \
  --cluster-config cluster-configs/default.yaml \
  --cluster-config cluster-configs/ronin.yaml
```

The jobs submitted to Slurm by running this command, as is, will fail. This is because commands being run by the various rules are expected to be available on the `PATH` and this is usually not the case on an HPC cluster. Snakemake provides the ability to execute jobs within a singularity container and also to utilise conda for setting up software environments in which rules can be executed. Together, this provides an opportunity for establishing long-term reproduciblity in the execution of a workflow by placing a conda environment inside a Singularity container.

Conda environments are specified, per rule, using the `conda` directive associated with a rule. For the purpose of this workshop, each rule uses the exact same conda environment file `envs/tutorial.yml` like so:

```
rule fastqc_trimmed:
  input:
    ...
  output:
    ...
  conda:
    "envs/tutorial.yml"
  shell:
    """
    ...
    """
```

Take a look at the `envs/tutorial.yml`. What versions of `trimmomatic` and `bwa` are being used?

`trimmomatic`: 0.36 `bwa`: 0.7.17

Are these the latest version of currently available through Anaconda? Hint: search https://anaconda.org/

As of 14th Aug 2019, the latest versions of these two tools are:

`trimmomatic`: 0.39 (https://anaconda.org/search?q=trimmomatic) `bwa`: 0.7.17 (https://anaconda.org/search?q=trimmomatic)

Putting together the above mentioned slurm execution backend profile, cluster configuration files and conda environment file, we now have the basic command for submitting jobs to the slurm queue and to have those jobs executed within a singularity container using a conda software environment. However, before we start submitting jobs, it's a good idea

to get the conda environment setup beforehand. This is bacuase conda is a notorious resource hog, so we want to submit the conda environment creation step to the cluster instead of it running on the head node.

Lets setup the conda environment(s) first

```
module load \
  singularity-3.2.1-gcc-5.4.0-tn5ndnb

# Create the conda environment(s), ahead of running the workflow, by \
    submitting a job to slurm
# This will take ~5mins
sbatch --job-name 'conda-env_setup' --mem 2G --time 30:00 --wrap \
  'snakemake --use-singularity --use-conda --create-envs-only'
```

Once the conda environment setup job completes successfully, we are now ready to run our workflow and have it submit jobs for execution by the cluster. This is most appropriately done on the head node rather than submitting it as a job to slurm. This is because the Snakemake process is essentially monitoring the slurm cluster for job status changes. As jobs complete sucessfully, Snakemake submits new jobs as and when their input dependencies have been satisfied. Because of this, the snakemake process is active for long as the workflow takes to complete - this could be many days or weeks depending on the size of the workflow and data set.

We're now ready to run the workflow and have the jobs submitted to the slurm queue:

```
snakemake \
  --profile profiles/slurm \
  --cluster-config cluster-configs/default.yaml \
  --cluster-config cluster-configs/ronin.yaml \
  --use-singularity \
  --use-conda
```

The provided profile, changes a few of the default values for some of Snakemake's commandline arguments.

Modify/extend the Workflow

Your own Workflow

Snakemake Troubleshooting

Snakemake Install

If you have a broken or incomplete snakemake installation, try the following steps to fix things:

```
# deactivate the snakemake conda environment if it is already active
conda deactivate

# Delete the snakemake conda environment
conda env remove --name snakemake
```

Now try reinstalling snakemake.

Conda Software Environment Setup

If your job failed or timed out, you will need to re-run conda software environment setup job again. However, you may first need to release the Snakemake lock which protects you from running multiple instances of the same workflow at the same time:

```
snakemake \
  --unlock
```

To ensure Snakemake starts with a clean slate, delete the "hidden" `.snakemake` directory:

```
rm -rf .snakemake
```

Getting Going After a Disconnect

If you find that your connection to the server has been dropped, you can get yourself going again using this convienient block of commands:

```
# Load the required software modules
module load \
  miniconda3-4.6.14-gcc-5.4.0-kkzv7zk \
  singularity-3.2.1-gcc-5.4.0-tn5ndnb

# Activate the snakemake conda environment and integrate shell \
    autocompletion into bash
conda activate snakemake
```

```
complete -o bashdefault -C snakemake-bash-completion snakemake

# Move to the correct directory location
cd /shared/${USER}/snakemake-tutorial
```

Introduction to Nextflow

Primary Author(s):
Radosław Suchecki rad.suchecki@csiro.au

Contributor(s):
Nathan S. Watson-Haigh nathan.watson-haigh@adelaide.edu.au

Key Learning Outcomes

After completing this module the trainee should be able to:

- Install Nextflow and execute an existing Nextflow workflow locally
- Modify the workflow to allow its execution on a compute cluster
- Write simple Nextflow process definitions and connect them with channels
- Apply operators to transform items emitted by a channel
- Leverage Nextflow's implicit parallelisation to process multiple data chunks independently

Resources Required

For the purpose of this training you need access to:

- A compute cluster with the `module` command available to you for loading software
- https://sylabs.io/singularity/Singularity - available as a module on the above cluster
- https://www.anaconda.com/distribution/conda - available as a module on the above cluster

Tools Used

Nextflow
https://nextflow.io

Graphviz
https://www.graphviz.org

Useful Links

Nextflow Documentation
https://www.nextflow.io/docs/latest/index.html

Nextflow Patterns
http://nextflow-io.github.io/patterns/

Slurm Documentation
https://slurm.schedmd.com/documentation.html

Introduction

Setting Up Your Environment

For the purpose of the workshop we will be working on the head node of an HPC cluster running Slurm. This is the most likely infrastructure that fellow bioinformaticians already find themselves using on a regular basis. We also assume that the cluster provides the `module` command for you to load software and the modules Java and Singularity are available to use.

The execution of the Nextflow workflow will take place on the cluster head node with jobs being submitted to Slurm for queuing and processing. From the head node, Nextflow will monitor the submitted jobs for their completion status and submit new jobs as dependent jobs complete successfully.

Connect to the Cluster Head Node

First up, lets connect to the head node of the HPC cluster using `ssh`.

See your local facilitator for connection details. You should have one user account per person.

Install nextflow

```
# Load the Java module on your cluster
# If it's unavailable contact the cluster sysadmin
module load openjdk-1.8.0_202-b08-gcc-5.4.0-sypwasp

# Download and install nextflow executable
curl -s https://get.nextflow.io | bash

# You should now be able to run it
./nextflow help
```

The installation should have placed the executable in your working directory. It is preferable to move the executable to a directory accessible via `$PATH`, to be able to run `nextflow` rather than having to remember to type the full `/path/to/nextflow` each time you want to run it.

Depending on the system this may suffice:

```
mkdir -p $HOME/bin
mv ./nextflow $HOME/bin
```

You should now be able to run `nextflow` without specifying the location of the binary.

Let's see if it works by running a script which is nextflow's take on 'hello world'.

Hello (nextflow) world!

```
nextflow run rsuchecki/hello
```

Nextflow will pull the `rsuchecki/hello` GitHub repository and run its main script.

We are relying on nextflow's integration with git and git registries. The **alternative** would be to

```
git clone https://github.com/rsuchecki/hello.git
nextflow run hello/main.nf
```

In which case the location of the cloned repository will be different to the one used by nextflow. You will also not have access to nextflow-git integration functionality.

Where do we find the local copy of `hello`? Hint: try `nextflow` commands related to pipeline sharing, such as `list` and `info`.

```
# List local clones of remote repositories
nextflow list
# Get detailed info about a repository
nextflow info hello #or nextflow rsuchecki/hello
```

```
project name: rsuchecki/hello
repository  : https://github.com/rsuchecki/hello
local path  : /home/rad/.nextflow/assets/rsuchecki/hello
main script : main.nf
revisions   :
* master (default)
  mybranch
  slurm
  testing
  v1.1 [t]
  v1.2 [t]
```

For now, we are mostly interested in the local path to the repository, the file name of the main script and its contents, which we will discuss next.

While waiting for others to catch up, why not have a look into how you would go about pulling and removing local clones of remote repositories using nextflow.

```
# remove local copy of rsuchecki/hello
nextflow drop hello
# pull rsuchecki/hello from remote without running the main script
nextflow pull rsuchecki/hello
```

What revisions (git branches or tags) are available for `nextflow-io/hello`? How would you run a specific revision?

```
# Available revisions
nextflow info hello
# Using -r/-revision, pointing to a listed tag or branch
nextflow run hello -revision v1.1
```

Nextflow basics

Processes and channels

- *process* – a wrapper for a language-agnostic script which ensures isolation of the executed code.
- *channel* – an asynchronous[1] FIFO queue which facilitates data flow to/from/between processes by linking their outputs/inputs.

[1] send operation completes immediately, receiving stops the receiving process until the message has arrived

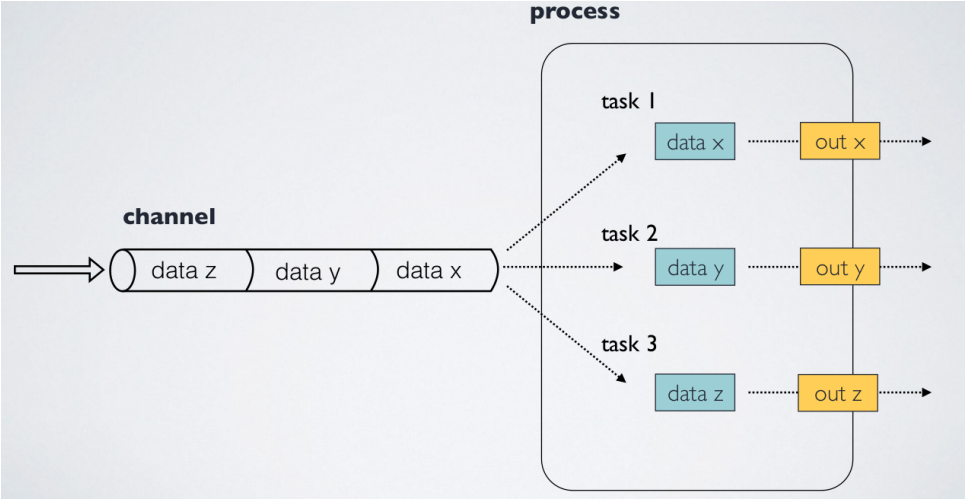

Figure 6: Nextflow building blocks: a *channel* "feeding" a processes. A *task* is an instance of a process. An isolated task is created for each emission (data chunk) from the input channel. Credit: Evan Floden

The main script

A nextflow script file name can be anything but in most cases it is best to stick to the default `main.nf`. The main script for the 'hello' example is as follows:

```
1  #!/usr/bin/env nextflow
2  echo true
3
4  cheers = Channel.from 'Bonjour', 'Ciao', 'Hello', 'Hola'
5
6  //setting default value, to be modified at runtime
7  params.world = 'world'
8
9  process sayHello {
10   input:
11     val x from cheers
12   script:
13     """
14     echo '$x $params.world!'
15     """
16 }
```

A channel called `cheers` is created and emits each of the listed strings separately. A separate instance of the process `sayHello` is executed for each emission.

The contenet of the above script can be broken down as follows:

- The shebang line (line 1) is optional.

- Setting `echo true` will output `stdout` of (every) process to the terminal - not advised for real world applications.
- `Channel.from(some_list)` creates a channel emitting the list elements one by one.
- Process definition (lines 6-13)
  - Input block (lines 7-8)
  - Script block (lines 9-12)
- The `$x` in the script block is a nextflow variable local to the process, not a bash variable.
- Indentation is inconsequential.

In addition process directives could be inserted above the input block.

Hello HPC!

The nextflow hello example shown us how the `sayHello` process was executed separately for each input string as a separate *task*, but all the tasks were executed locally on our cluster's head node. We would now like each task to be submitted as a batch job for execution on one of the compute nodes.

```
1 nextflow run rsuchecki/hello -revision slurm
```

This is the modified version of the `main.nf` script. Submission to Slurm was achieved by adding `executor 'slurm'` directive to the process definition.

```
1  #!/usr/bin/env nextflow
2  echo true
3
4  cheers = Channel.from 'Bonjour', 'Ciao', 'Hello', 'Hola'
5
6  //setting default value, to be modified at runtime
7  params.world = 'world'
8
9  process sayHello {
10   executor 'slurm'
11
12   input:
13     val x from cheers
14   script:
15     """
16     echo "$x $params.world from \$HOSTNAME on Slurm!"
17     """
18 }
```

You might also have noticed that we have modified the script block so that the messages printed to the terminal include the name of the compute node on which a given task is executed.

Note the difference between how nextflow variables (`$x`,`$params.world`) and bash variables (`$HOSTNAME`)are included in the script block. There are alternative ways of including variables in scripts for execution by nextflow processes which may be more convenient if your script contains multiple special characters.

Hello task caching!

When the pipeline is launched with the `-resume` option, any attempt to execute already executed process with the same inputs, will cause the process execution to be skipped, producing the stored data as the output.

In this toy example we do not specify any outputs but the 'hello' messages printed to the terminal reflect this behaviour.

```
1 nextflow run rsuchecki/hello -revision slurm -resume
```

To avoid unintentionally re-computing long running tasks you may consider always running your pipelines with `-resume` and only omitting it on rare occasions when you want to re-compute the results even though inputs have not changed.

https://www.nextflow.io/docs/latest/process.html#cache

Hello command line options

Single-dashed options are reserved for nextflow engine (`-resume`, `-revision`, `-ansi-log false` etc). The double-dashed options are all yours and you are free to use them for your workflow. When you `nextflow run some_script.nf --foo bar`, the value of the parameter ('bar') will be accessible in `main.nf` as `params.foo` and within a script block as `$params.foo`.

In the 'hello' example we use `params.world` which by default is set to 'word', so lets try to use an alternative string.

```
1 nextflow run rsuchecki/hello -revision slurm --world Mundo
```

Goodbye Hello

Nextflow facilitates but does not enforce separation of workflow logic from the configuration of compute and software environments as well as from other properties of the workflow. As such, you *could* get by developing nextflow workflows without worrying about that aspect – but you would be missing a lot in terms of flexibility, extensibility, portability and more

Nextflow looks for workflow configuration primarily in `nextflow.config` file, and additional config files can be included. Unsurprisingly the 'hello' example does not require much configuration, we would also like to crunch some real, albeit small, data.

This is mostly symbolic

```
1 nextflow drop rsuchecki/hello
```

Let's have a play with a slightly more practical workflow.

Example workflow

We are going to work with an example Nextflow workflow to demonstrate how they are run, improve your understanding of *processes* and *channels* and finally introduce *operators*, which are applied to channels to shape and direct flowing data.

This example workflow consists of the following steps:

- Running FastQC across the raw reads
- Aggregating the raw read FastQC reports using MultiQC
- Performing adapter, quality, and read length filtering using Trimmomatic
- Running FastQC across the QC'd reads
- Aggregating the QC read FastQC reports using MultiQC
- Indexing the reference FASTA file
- Performing a `bwa-mem` read alignment

Although not necessary for simply running the pipeline, in the training context it makes sense to start by cloning the workflow repository and moving to the directory.

```
mkdir -p /shared/${USER}/nextflow-tutorial
cd /shared/${USER}/nextflow-tutorial
git clone \
    https://github.com/csiro-crop-informatics/nextflow-embl-abr-webinar.git \
    example_workflow
cd example_workflow
git checkout noslurm
git branch
```

This time, in addition to `main.nf` we have a separate script which downloads the required data sets, which include a small reference FASTA file and 16 pairs of FASTQ files, each for a different bread wheat accession.

```
nextflow run setup_data.nf
```

If successful, we could now try to run the workflow...

```
nextflow run main.nf
```

This is expected to fail.
Unless all the software required by the pipeline is available on the `$PATH`, which we don't expect, the pipeline should terminate with an error. The output information may help you identify the cause. Try to relate the error message to the relevant section of the main script (`main.nf`).

Which process has failed? What was the underlying cause? The cause was likely "`command not found`" and it may have been any of the processes for which the software tool is not available. Example:

```
Error executing process > 'fastqc_raw (Xiaoyan)'

Caused by:
  Process `fastqc_raw (Xiaoyan)` terminated with an error exit \
      status (127)

Command executed:

  fastqc --quiet --threads 1 *

Command exit status:
  127

Command output:
  (empty)

Command error:
  .command.sh: line 2: fastqc: command not found

Work dir: \
    /tmp/nextflow-embl-abr-webinar/work/15/505ea816d2411e68ea253ee126c181
```

There are two main issues with executing this workflow as is,

1. Third-party software tools have not been made available to the workflow.
2. We are trying to run the entire workflow on the cluster's head node.

There are different ways in which these issues could be addressed, for example using process *directives* at the top of each process definition. Depending on your cluster configuration this could be for example:

```
process foo {
  executor 'slurm'
  module 'samtools/1.9'
  //further code omitted
```

This is a perfectly valid syntax, which can be convenient, particularly during pipeline development, but for more portable workflows it is preferable to keep compute and software environment configuration separate from pipeline logic – in simple terms not in the workflow script (`main.nf`).

The config file(s) and profiles

Workflow configuration belongs in `nextflow.config` file. Transferring the above mention *directives* from process definitions in `main.nf` to `nextflow.config` would make things slightly better, e.g.

```
#nextflow.config

process.executor = 'slurm'
process.module = 'samtools/1.9'
```

or using the preferred syntax

```
process {
  executor = slurm
  module = 'samtools/1.9'
}
```

This is however still a bit rigid.

- You may be developing your pipeline on a local machine or a server where software modules are not available.
- If developing directly in the cluster environment, you may prefer your quick test runs to happen either on the head node or in an interactive session you are using, rather than always having jobs submitted to sit in the always-busy cluster queue.

Nextflow enables the definition of *profiles* which make it easy to run a workflow with different configuration settings, including, but not limited to executors and software environment.

For our pipeline we have defined several *profiles*, which allow us to execute the logic in `main.nf` while providing the required software either by creating a `conda` environment or by using Docker of Singularity containers where the conda environment has already been captured.

Relevant profiles

Identify the profile definitions in `nextflow config`. The ones most immediately relevant are:

```
profiles {
  //SOFTWARE
  conda {
    process {
      conda = "$baseDir/conf/conda.yaml"
    }
  }
  singularity {
    process {
      container = \
          'shub://csiro-crop-informatics/nextflow-embl-abr-webinar'
    }
    singularity {
      enabled = true
      autoMounts = true
      cacheDir = "singularity-images"
    }
  }
}
```

As you can see, Nextflow makes it really easy to define software environment via Singularity or Conda[2].

Given that Singlularity is available on our cluster, let's start by using that profile, as the most robust way of setting up the software environment.

We will need Singularity for nextflow to be able to pull the container image from Singularity Hub and run the containerised software. By default the pipeline will process reads for a single accession – our head node should be able to handle this.

```
# Load the Singularity module
# If it is unavailable contact the cluster sysadmin

module load singularity-3.2.1-gcc-5.4.0-tn5ndnb

# Run the workflow

nextflow run main.nf -profile singularity
```

This is sufficient when running a workflow locally, in an interactive session or on a standalone server. The next step is to get nextflow to make use of the HPC batch submission

[2]We also have a docker profile which you may find useful if you decide to run the workflow on your machine

system, to be able to run the full workflow without unleashing your sysadmins wrath.

Edit `nextflow.config`. Your task is to add a `slurm` profile which will set the appropriate executor.

What is your `slurm` profile configuration and where do you place it in `nextflow.config`?
The following code should be placed within the `profiles{}` block in `nextflow.config`.

```
slurm {
  process {
    executor = 'slurm'
  }
}
```

There are of course many setting that can and in some cases must be set – refer to executors section of Nextflow documentation[3]. For running real-life pipelines in a cluster environment you will also use directives [4] controlling the resources (`cpus, memory, time`) requested for each job. Other possibly relevant directives include `queue` and `scratch`.

Cluster run

To avoid running the workflow on our head node or in an interactive session, we will use the `slurm` profile you have defined[5]. As before, the software environment will be handled via the `singularity` profile. For that, we will need Singularity on the head node for nextflow to be able to pull the container image from Singularity Hub (we could also use a locally stored image). Singularity will also be required on the compute nodes which will run the individual tasks, but this should happen seamlessly if an appropriate module is loaded on the head node, otherwise the required module would also have to be specified in the workflow configuration files.

By default a single accession will be processed. You may use the `-resume` flag to avoid re-computing already existing results.

```
# Load the Singularity module on your cluster
# If it is unavailable contact the cluster sysadmin

module load singularity-3.2.1-gcc-5.4.0-tn5ndnb

# Run the workflow

nextflow run main.nf -profile slurm,singularity -resume
```

[3] https://www.nextflow.io/docs/latest/executor.html
[4] https://www.nextflow.io/docs/latest/process.html#directives
[5] If you are struggling and can't get help, try: `git stash && git checkout workshop`

Under the hood

If you think you are ready to look under the hood and try to work out how nextflow stages process inputs, wraps process script blocks and submits them to the cluster, here is a start.

```
# Remove the work directory to limit the number of task directories \
    to look at
rm -r work
# Re-run for a single sample
nextflow run main.nf -profile slurm,singularity
# Take a peak
ls -la work/ | less
# or
tree -ah work/ | less
```

Each task is executed in a separate directory and every abbreviated hash displayed in the terminal can be related to a specific sub-directory of `./work`,
such as `work/d2/c4517b0a81f61ceca29ec355ddeaa6/` in which you may find

```
# NF generated files
.command.begin
.command.err
.command.log
.command.out
.command.run
.command.sh
.command.trace
.exitcode

# Output file
H45.bam

# Symlinks to input files
H45_R1.paired.fastq.gz
H45_R2.paired.fastq.gz
reference.fasta.gz.amb
reference.fasta.gz.ann
reference.fasta.gz.bwt
reference.fasta.gz.pac
reference.fasta.gz.sa
```

Identify and investigate hidden file (starting with dot) containing the executed script and the one containing cluster and container handling.

Cluster run - all accessions

We have successfully submitted workflow to the cluster.

To be sure, feel free to re-run it again (and again, and again...) with `-resume` to avoid wasting CPU cycles.

```
nextflow run main.nf -profile slurm,singularity -resume
```

If all went well, the workflow successfully processed a single accession, let's have a closer look at the script to better understand how it handles the inputs before we proceed to run it on all the accessions.

In `main.nf` we create a channel which reads pairs of FASTQ files from a sub-directory of the `./data`. We then apply some operators.

```
Channel.fromFilePairs("data/${region}/*_R{1,2}.fastq.gz")
  .take ( params.take == "all" ? -1 : params.take )
  .into { readPairsChannelA; readPairsChannelB }
```

1. Identify the two operators, refer to nextflow documentation[a] as required and explain the purpose of each of the two operators.
`.take(`n`)` limits the number of emissions from the channel to the first n items.
`.into{` $ch_1; ch_2; ...; ch_n$ `}` creates channels $ch_1, ch_2, ..., ch_n$ and connects source channel to the newly created channels, so that every emission is sent through each new channel.

2. How can you run the workflow for more than one accession? How about all of them? Recall that workflow parameters use double-dash syntax. Run the relevant commands.

```
nextflow run main.nf -profile slurm,singularity -resume --take 2
nextflow run main.nf -profile slurm,singularity -resume --take all
```

[a] https://www.nextflow.io/docs/latest/operator.html

Monitoring your jobs on our cluster

You can monitor your job(s) in the slurm queue using the slurm command `squeue`:

```
squeue --user ${USER}
```

For convenience you are also provided with the `sq` function which produces nicer output and by default only shows your own jobs:

```
sq

# Someone elses jobs
sq --user ${SOMEONE_ELSE}
```

If you want to see all jobs in the queue:

```
squeue
```

For an optional exercise you may try to re-run the workflow with `conda`. For that, you'll need to find and load a conda module before re-running the workflow with appropriate profile. Don't forget to use the `-resume` flag.

```
# Find the appropriate module name
module av -l 2>&1 | grep conda

# Load the module
module load miniconda3-4.6.14-gcc-5.4.0-kkzv7zk

# Run with conda
nextflow run main.nf -profile conda,slurm --take all -resume
```

If you remembered to use `-resume`, why do you think it appeared to not make a difference?

We have switched from singularity to conda so the software environment has changed.

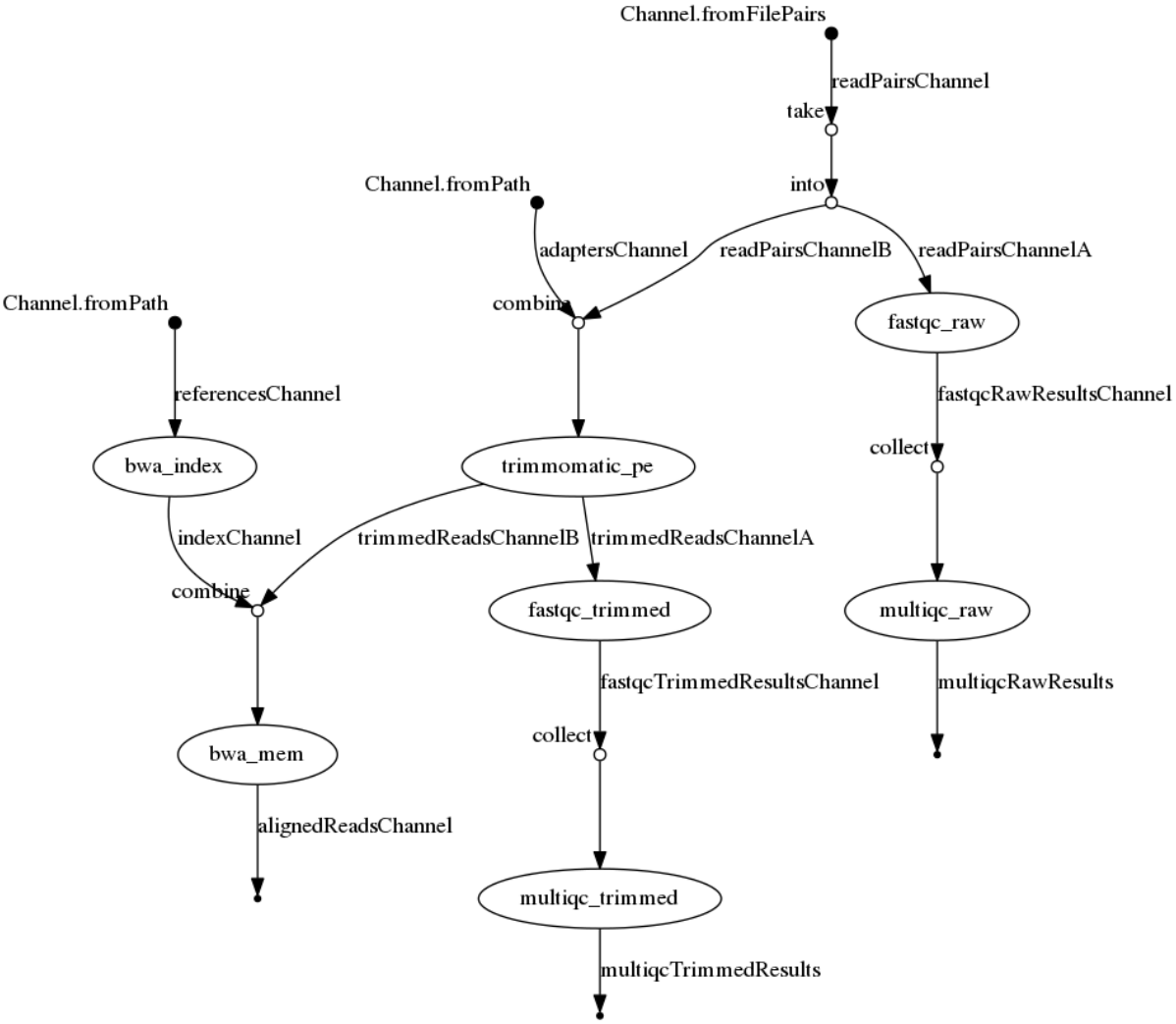

Figure 7: The example workflow

Investigate `main.nf` alongside Figure 7.

Which nextflow operators[a], in addition to the previously discussed, are used and for what purposes?

The `.combine()` operator outputs all combinations of items emitted by two channels. This results with a downstream process to be executed for each such combination. So e.g. `bwa_mem` will be executed for $(reference, accession_1), (reference, accession_2), .., (reference, accession_n)$.

The `.collect()` operator collects all the items emitted by a channel returns the resulting `List` as a single emission. This is required e.g. if a process needs to be executed once with all the samples as input.

[a] https://www.nextflow.io/docs/latest/operator.html

Workflow outputs

We now now each task is nicely isolated in a separate sub-directory under `work`, but how do I find my results? Was it `work/a7:fc9339a827fb4b34d2408e1c3ee29c` or maybe `work/3c:8fdf958e96b448ecb83bd7806af382`? This should be handled by applying the `publishDir` directive[6] to selected processes. As with other directives, this can be included at the top of the process block or in a configuration file using process selectors[7] to apply the directive to one or more relevant process. To keep things tidy-ish, we define the publishing of the outputs in a separate file which we `includeConfig 'conf/publish.config'` in `nextflow.config`.

In `conf/publish.config` we only really use the `withName` selectors. The alternative `withLabel` selectors are convenient e.g. when outputs of multiple processes are to be gather in one location, in which case we attach the same `label` to each of those processes.

Modify/extend the workflow

Edit `main.nf`. Your task is to add a process which will merge the bam files produced by the `bwa_mem` process.

How do you ensure that **all** BAM files end up in the same instance of your process? Using the `.collect()` operator. Demonstrate your process definition to your facilitator.

```
process bam_merge {
  input:
    file('*.bam') from alignedReadsChannel.collect()

  script:
  """
  samtools merge ${params.take}_accessions_megred.bam *.bam
  """
}
```

Where can we find the merged BAM file? Can you publish it to a human-readable location? Hint: only declared outputs can be published.

[6] https://www.nextflow.io/docs/latest/process.html#publishdir
[7] https://www.nextflow.io/docs/latest/config.html#process-selectors

Modify your merge process to allow samtools to use 2 cpus with `--threads 2`, don't forget to modify your process configuration to request 2 cpus per task.

Your own workflow (TODO: replace with variant calling?)

It is time to have a go at your own pipeline. Since we have some inputs and configuration files at hand, you can start a `own.nf` script file in the current directory and read the input files from `./data`.

The simple pipeline should include the following:

- Code for reading FASTQ read files from `./data` individually (i.e. not as pairs) into a channel.
- A process which will take a read file, count the reads and output the file name alongside the read count.
- A way of aggregating the individual count files into a single csv file. This could be done in another process or using an operator.

There are different ways of approaching the exercise, here is an example solution. For comparison, we demonstrate the aggregating step both as a process and using the `collectFile()` operator.

```
readsChannel = Channel.fromPath("data/**.fastq.gz")

process countReads {
  input:
    file fastq from readsChannel

  output:
    file '*' into countsChannel1, countsChannel2

  """
  echo -ne "${fastq}," > count
  zcat $fastq | paste - - - - | wc -l >> count
  """
}

process aggregate {
  publishDir params.outdir

  input:
    file '*.count' from countsChannel1.collect()

  output:
    file '*.csv'

  """
  cat *.count > counts_from_process.csv
  """
}

countsChannel2.collectFile(name: 'counts_from_operator.csv', \
    storeDir: params.outdir)
```

Troubleshooting

Disconnected from the cluster?

Missing modules - new shell session?

Make sure all the required modules are loaded.

```
# Java - essential for nextflow
module load openjdk-1.8.0_202-b08-gcc-5.4.0-sypwasp

# Singularity - our go to system for providing software for the example \
    workflow
module load singularity-3.2.1-gcc-5.4.0-tn5ndnb


# If using conda
module load miniconda3-4.6.14-gcc-5.4.0-kkzv7zk
```

Space for Personal Notes or Feedback

Space for Personal Notes or Feedback

Space for Personal Notes or Feedback