Implementing Scalable Bioinformatic Workflows in Snakemake and Nextflow

Nathan S. Watson-Haigh
Radosław Suchecki

Across Australia Aug/Sep 2019

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 should have one user account per person.

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 to integrate conda into bash
#####
conda init bash
. ~/.bashrc
#####

# Install snakemake by submitting a job to slurm - conda is a resource
# hog so your HPC sysadmins will like you for doing this
# This might take 10-20mins
sbatch --job-name 'snakemake-install' --mem 4G --time 30:00 --wrap \
 "conda create \
```

```
  --name snakemake \
  --channel bioconda --channel conda-forge \
  --yes \
  snakemake=${SNAKEMAKE_VERSION}"
```

The above `conda create` command, submitted to slurm, will take 10-20mins to complete.

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}
```

Once your job completes successfully, snakemake installation is complete. 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.

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
2  mkdir -p /shared/${USER}/my_hello_world
3  cd /shared/${USER}/my_hello_world
4
5  # Create the Snakefile with the relevant contents using this heredoc
6  cat >Snakefile <<EOF
7  rule hello_world:
8    output:
9      "Hello/World.txt",
10   shell:
11     """
12     echo "Hello, World!" > Hello/World.txt
13     """
14 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 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?

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

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

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

# Explicityly 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
```

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}
    """
```

Example Workflow

We are going to work with an example Snakemake workflow to demonstrate how they are run before we get stuck into writing our own. 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
- Index the reference FASTA file
- Perform a `bwa-mem` read alignment

Lets get the example workflow:

```
# Clone the repository
git clone \
    https://github.com/UofABioinformaticsHub/2019_EMBL-ABR_Snakemake_webinar \
    ./example_workflow

cd example_workflow/snakemake-tutorial
```

The most basic of Snakemake commands will execute the workflow, running jobs on the

head node (your cluster sysadmin will probably growl at you). Lets be nice, and just do a `dryrun` for now:

```
snakemake \
  --dryrun
```

Using the Snakemake help, how do you get Snakemake to print the shell commands it will execute for each job?

Modify/extend the Workflow

Your own Workflow

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):

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 hello
```

Nextflow will pull the `nextflowio/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/nextflow-io/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.

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

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.

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

Nextflow basics

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.

The main script

A nextflow script file name can be anything but for some purposes 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  process sayHello {
7    input:
8      val x from cheers
9    script:
10     """
11     echo '$x world!'
12     """
13 }
```

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.

Nextflow looks for workflow configuration files, primarily in `nextflow.config` file, and additional config files can be included. Unsurprisingly the 'hello' example does not require much configuration, so let's look at a slightly more practical workflow.

Example workflow

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

```
git clone \
    https://github.com/csiro-crop-informatics/nextflow-embl-abr-webinar.git
cd nextflow-embl-abr-webinar
git checkout workshop
```

TODO: prepare a `workshop/noslurm` branch lacking slurm config details, which the participants would check out at this stage.

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. Spend a few minutes relating the error message content to the relevant section of the main script (`main.nf`).

Which process has failed? What was the underlying cause?

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 - i.e. not in the workflow script (`main.nf`).

The config file(s) and profiles

As mentioned earlier, 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 alternative 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 server and where software modules are not available. If developing directly in the cluster environment, you may prefer to your quick test runs happen either on the head node or in an interactive session you are using, rather than jobs being submitted to sit in the always-busy cluster queue.

Nextflow enables the definition of *profiles* which enables you 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 (we also have a docker profile).

You could now run the workflow with Singularity on the head node (generally not advised), or by starting an interactive session on one of the compute nodes.

```
srun --pty bash
```

and once you get onto one of the compute nodes 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.

```
# 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

# Release the resources - leave the interactive session.

```

```
12 exit
```

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.

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`?

There are of course many setting that can and in some cases must be set - refer to executors section of Nextflow documentation. For running real-life pipelines in a cluster environment you will also use directives 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. 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.

```
1 # Load the Singularity module on your cluster
2 # If it is unavailable contact the cluster sysadmin
3
4 module load singularity-3.2.1-gcc-5.4.0-tn5ndnb
5
6 # Run the workflow
7
```

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

Under the hood

If you think you are ready to look under the hood and try to work out how nextflow handles all the 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
tree -ah work/ | less
```

Each task is executed in a separate directory and an 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. 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 as required and explain the purpose of each of the two operators.

2. How can you run the workflow for more than one accession? How about all of them?

3. Investigate `main.nf` to identify the processes which consume `readPairsChannelA` and `readPairsChannelB`.

Q????

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.

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

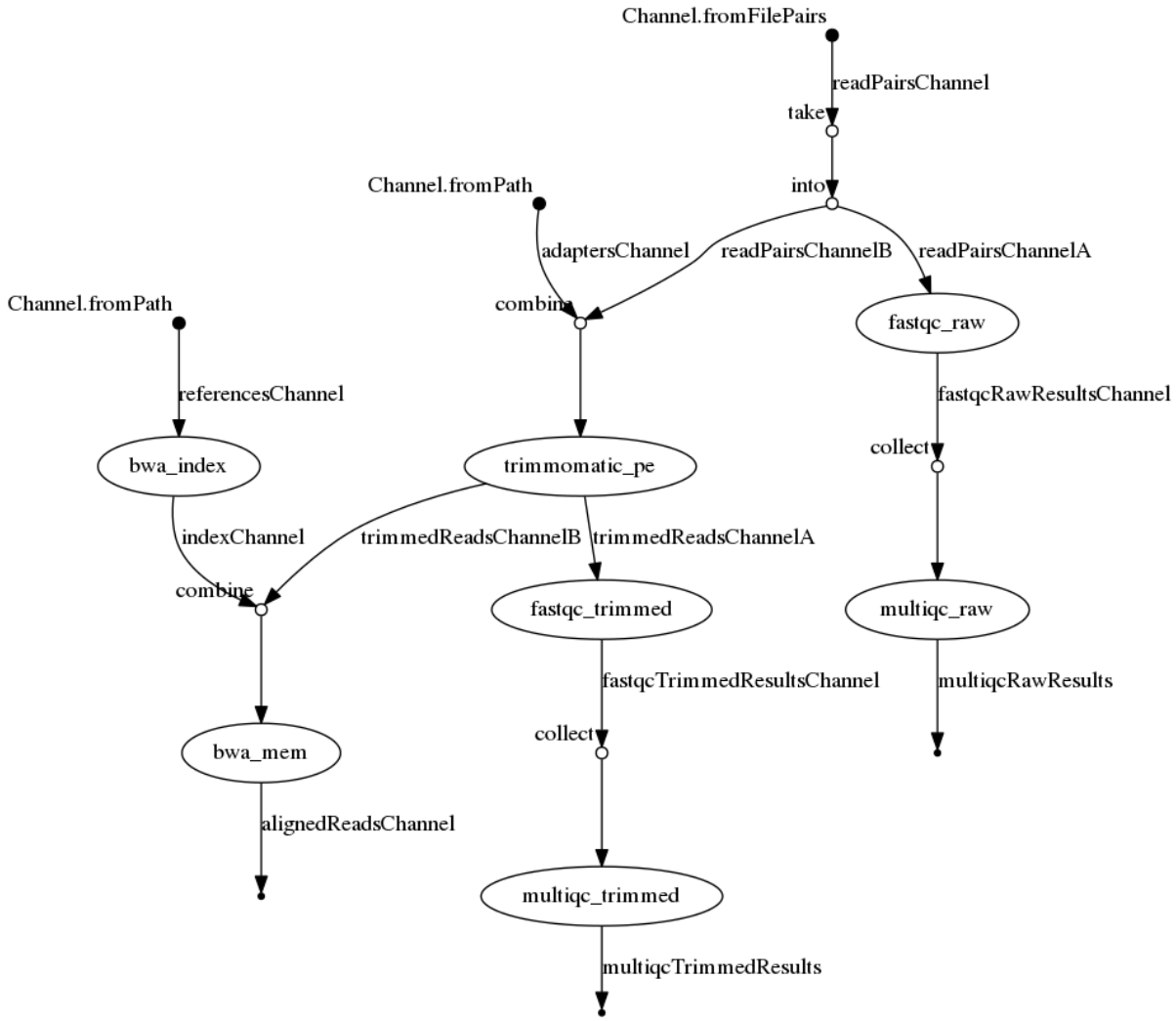

Figure 1: The example workflow

Investigate `main.nf` alongside Figure 1.

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

Modify/extend the workflow

Option 1
Edit `main.nf`. Your task is to add another multiqc process which will summarize all fastqc results together, i.e. those from raw and from the trimmed data.

Option 2 (advanced)
Add a process which will take all raw reads and all paired reads for a given sample, count the number of raw paired reads and the number of paired reads remaining after the trimming.

Your own workflow

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.

Key concepts to cover

- channels
- operators
- processes
- directives

Under the hood

Parametrisation

- single dash params
- double dash params
- environmental variables

Note that currently the default behaviour of Nextflow is to re-run an entire workflow

unless `-resume` option is specified at run-time, in which case a previously executed process is not re-run if all its inputs remain unchanged.

additional exercises

1. `publishDir`
2. process selectors (optional)

Questions and Answers

It is useful to maintain the answers to questions together in the same LaTeXsource. By using the `\begin{answer} ... \end{answer}` environment we can have the enclosed answers excluded from the trainee's version of the handout while including it in the trainer's version of the handout.

We can ask a simply question to test if the trainee is understanding what they are doing.

Space for Personal Notes or Feedback