
Implementing Scalable Bioinformatic Workflows in Snakemake and Nextflow

Nathan S. Watson-Haigh
Radosław Suchecki

Licensing

This work is licensed under a Creative Commons Attribution 3.0 Unported License and the below text is a summary of the main terms of the full Legal Code (the full licence) available at <http://creativecommons.org/licenses/by/3.0/legalcode>.

You are free:

- to copy, distribute, display, and perform the work
- to make derivative works
- to make commercial use of the work

Under the following conditions:

Attribution - You must give the original author credit.

With the understanding that:

Waiver - Any of the above conditions can be waived if you get permission from the copyright holder.

Public Domain - Where the work or any of its elements is in the public domain under applicable law, that status is in no way affected by the license.

Other Rights - In no way are any of the following rights affected by the license:

- Your fair dealing or fair use rights, or other applicable copyright exceptions and limitations;
- The author's moral rights;
- Rights other persons may have either in the work itself or in how the work is used, such as publicity or privacy rights.

Notice - For any reuse or distribution, you must make clear to others the licence terms of this work.



Contents

Licensing	3
Contents	4
Workshop Information	5
The Trainers	6
Providing Feedback	7
Document Structure	7
Introduction to Snakemake	9
Key Learning Outcomes	10
Resources Required	10
Useful Links	10
Setting Up Your Environment	11
Snakemake basics	13
Example Workflow	20
Modify/extend the Workflow	24
Your own Workflow	24
Snakemake Troubleshooting	24
Introduction to Nextflow	27
Key Learning Outcomes	28
Resources Required	28
Useful Links	29
Introduction	30
Setting Up Your Environment	31
Nextflow basics	33
Example workflow	37
Modify/extend the workflow	46
Your own workflow (TODO: replace with variant calling?)	47
Troubleshooting	48
Space for Personal Notes or Feedback	49

Workshop Information

The Trainers

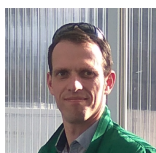


Dr. Nathan S. Watson-Haigh

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:

```
1 R --no-save
2 library(plotrix)
3 data <- read.table("run_25/stats.txt", header=TRUE)
4 weighted.hist(data$short1_cov+data$short2_cov, data$lgth, breaks=0:70)
5 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 operates 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 queuing and processing. From the head node, Snakemake 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, let's 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 environment (https://snakemake.readthedocs.io/en/stable/getting_started/installation.html). As such, you need Anaconda3, usually available to you on your cluster via the module system.



```

1  # We use a specific version for reproducibility reasons
2  # Find the latest version: https://anaconda.org/search?q=snakemake
3  SNAKEMAKE_VERSION="5.5.4"
4
5  # Load miniconda
6  module load \
7      miniconda3-4.6.14-gcc-5.4.0-kkzv7zk
8
9  #####
10 # One-time commands to integrate conda into bash
11 #####
12 conda init bash
13 . ~/.bashrc
14 #####
15
16 # Install snakemake by submitting a job to slurm - conda is a resource
17 # hog so your HPC sysadmins will like you for doing this
18 # This might take 10-20mins
19 sbatch --job-name 'snakemake-install' --mem 4G --time 30:00 --wrap \
20 "conda create \

```

```

21 --name snakemake \
22 --channel bioconda --channel conda-forge \
23 --yes \
24 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 queue, or just your own job(s) using the slurm command `squeue`:

```

1 # All jobs in the queue
2 squeue
3
4 # Just your own jobs
5 squeue --user ${USER}

```

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

```

1 # Your own jobs
2 sq
3
4 # Someone elses jobs
5 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:



```

1 # Activate the newly created conda environment
2 conda activate snakemake

```

Integrate Snakemake autocompletion into bash:

```

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

```

Test if Snakemake is actually working:

```

1 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 (“dependencies”) 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

```
1 | snakemake
```

Try executing it for a second time:

```
1 | 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 explicitly ask for a “target”. This means the following are equivalent:

```
1 | # Not specifying a target will result in Snakemake executing the
2 | # first rule in the Snakefile ("hello_world" in this case)
3 | snakemake
4 |
5 | # Explicitly requesting the "hello_world" rule to be run:
6 | snakemake hello_world
7 |
8 | # Explicitly request the target filename to be created:
9 | 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 curly brace syntax within the `shell` directive.

So, rewrite the rule as follows:

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

`{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 equivalent:

```
1 # Delete output files of the implied target (the first rule in the \
   Snakefile)
2 snakemake --delete-all-output
3
4 # Delete output files of the explicitly defined "hello_world" rule:
5 snakemake --delete-all-output hello_world
6
7 # Delete output files of the explicitly requested target filename:
8 snakemake --delete-all-output Hello/World.txt
```

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

```
1 rule hello_world:
2     output:
3         "Hello/World.txt",
4     shell:
5         """
6         echo "Hello, World!" > {output}
7         """
8
9 rule hello_world_spanish:
10    output:
11        "Hola/World.txt",
12    shell:
13        """
14        echo "Hola, World!" > {output}
```

```
15 """
```

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:

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

We define a named wildcard called `cheer`, using curly 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:

```
1 # Not specifying a target will result in Snakemake executing the
2 # first rule in the Snakefile ("hello_world" in this case)
3 snakemake
4
5 # Explicitly requesting the "hello_world" rule to be run:
6 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 convenience 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:

```
1 rule all:
2     input:
3         "Bonjour/World.txt",
4         "Ciao/World.txt",
5         "Hello/World.txt",
6         "Hola/World.txt",
7
8 rule hello_world:
9     output:
10        "{cheer}/World.txt",
11    shell:
```

```
12 """
13 echo "{wildcards.cheer}, World!" > {output}
14 """
```

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:

```
1 # Not specifying a target will result in Snakemake executing the
2 # first rule in the Snakefile ("all" in this case)
3 snakemake
4
5 # Explicitly request the "all" rule
6 snakemake all
7
8 # Explicitly a cheer in Polish
9 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:

```
1 cheers = ['Bonjour', 'Ciao', 'Hello', 'Hola']
2
3 rule all:
4     input:
5         expand("{cheer}/World.txt", cheer=cheers),
6
7 rule hello_world:
8     output:
9         "{cheer}/World.txt",
10    shell:
11        """
12        echo "{wildcards.cheer}, World!" > {output}
13        """
```

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 rewrite our “Hello, World!” example to also translate the “World” word

we might think to do it like this:

```
1 | cheers = ['Bonjour', 'Ciao', 'Hello', 'Hola']
2 | worlds = ['Monde', 'Mondo', 'World', 'Mundo']
3 |
4 | rule all:
5 |     input:
6 |         expand("{cheer}/{world}.txt", cheer=cheers, world=worlds),
7 |
8 | rule hello_world:
9 |     output:
10 |         "{cheer}/{world}.txt",
11 |     shell:
12 |         """
13 |         echo "{wildcards.cheer}, {wildcards.world}!" > {output}
14 |         """
```

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:

```
1 | 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:
1  cheers = ['Bonjour', 'Ciao', 'Hello', 'Hola']
2  worlds = ['Monde', 'Mondo', 'World', 'Mundo']
3
4  rule all:
5      input:
6          expand("{cheer}/{world}.txt", zip, cheer=cheers, world=worlds),
7
8  rule hello_world:
9      output:
10         "{cheer}/{world}.txt",
11      shell:
12         """
13         echo "{wildcards.cheer}, {wildcards.world}!" > {output}
14         """
```

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 at 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:

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

```
3 cd /shared/${USER}/snakemake-tutorial
4
5 # Clone the example workflow repository from GitHub
6 git clone https://github.com/UofABioinformaticsHub/snakemake-tutorial ./
7 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:

```
1 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.

```
1 snakemake \
2 --dryrun
```



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

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.

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, let's 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 be piped into Graphviz' `dot` program to convert and create a PDF:

```
1 snakemake \
2   --dag \
3   | dot -Tpdf \
4   > dag.pdf
```

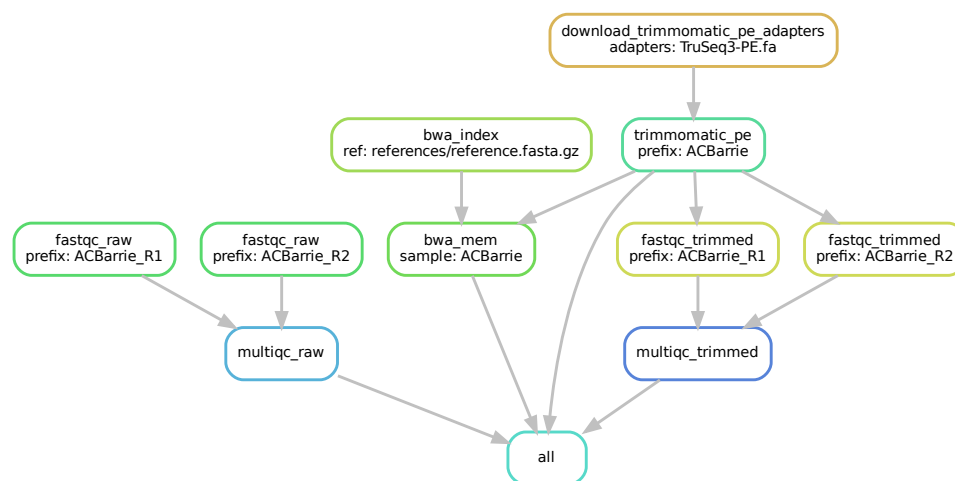


Figure 1: DAG of jobs where `accession` is `ACBarrie`.

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:

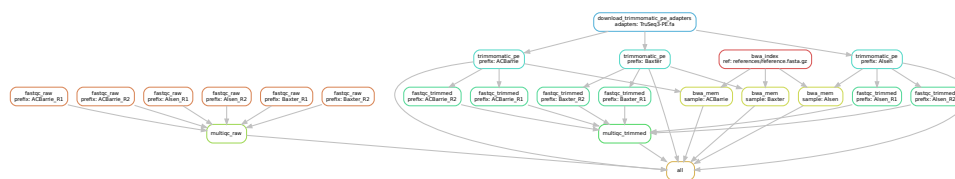


Figure 2: DAG of jobs where `accession` is `ACBarrie`, `Alsen` or `Baxter`.

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

```
1 | snakemake \  
2 |   --rulegraph \  
3 | | dot -Tpdf \  
4 | > rulegraph.pdf
```

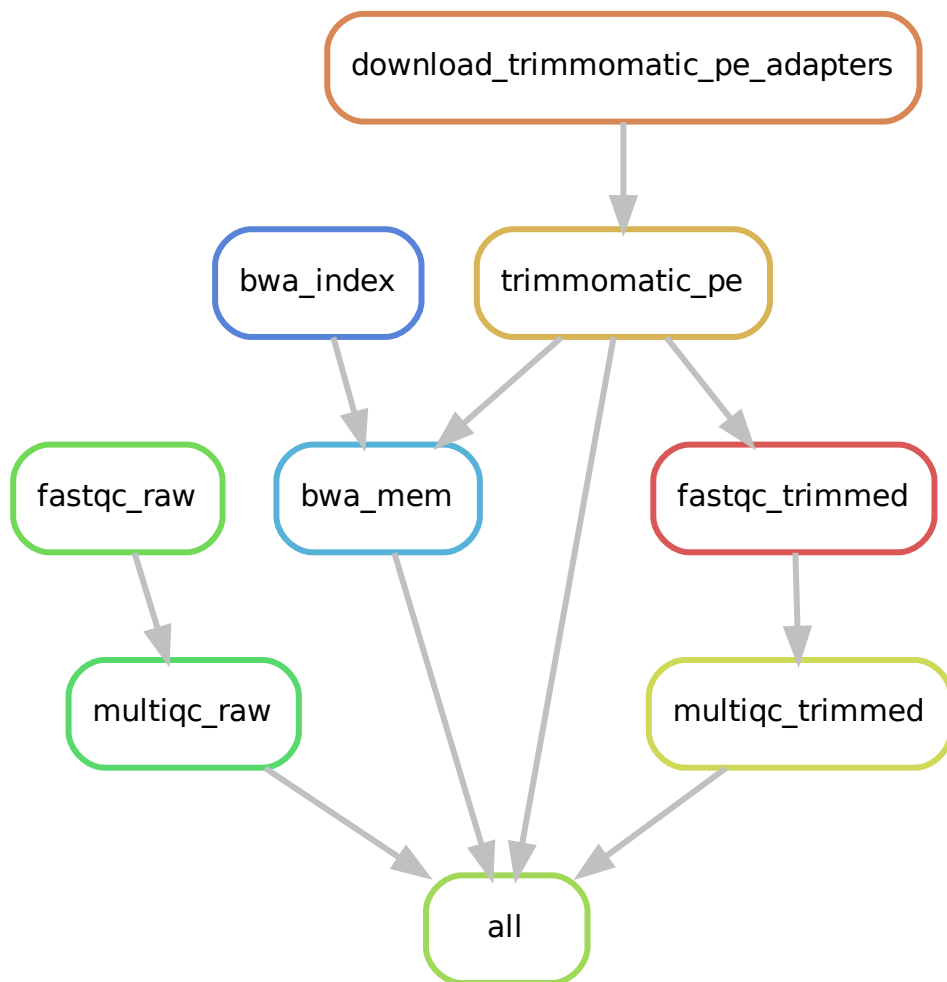


Figure 3: Graph of rules.

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 continue executing these resource hungry jobs on the head node, your cluster sysadmin will start to growl at you!

Snakemake does not

Snakemake provides the ability to execute jobs within a singularity container and to have a conda software environment set up within it. This provides an opportunity

```
1 module load \  
2     singularity-3.2.1-gcc-5.4.0-tn5ndnb  
3  
4 # Create the conda environment(s), ahead of running the workflow, by \  
5   submitting a job to slurm  
6 # This will take ~5mins  
7 sbatch --job-name 'snakemake-env_setup' --mem 2G --time 30:00 --wrap \  
8   'snakemake --use-singularity --use-conda --create-envs-only'
```

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:

```
1 # deactivate the snakemake conda environment if it is already active  
2 conda deactivate  
3  
4 # Delete the snakemake conda environment  
5 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:

```
1 | snakemake \  
2 | --unlock
```

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

```
1 | 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 convenient block of commands:

```
1 | # Load the required software modules  
2 | module load \  
3 |   miniconda3-4.6.14-gcc-5.4.0-kkzv7zk \  
4 |   singularity-3.2.1-gcc-5.4.0-tn5ndnb  
5 |  
6 | # Activate the snakemake conda environment and integrate shell \  
   |   autocompletion into bash  
7 | conda activate snakemake  
8 | complete -o bashdefault -C snakemake-bash-completion snakemake  
9 |  
10 | # Move to the correct directory location  
11 | 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



```
1 # Load the Java module on your cluster
2 # If it's unavailable contact the cluster sysadmin
3 module load openjdk-1.8.0_202-b08-gcc-5.4.0-sypwasp
4
5 # Download and install nextflow executable
6 curl -s https://get.nextflow.io | bash
7
8 # You should now be able to run it
9 ./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:



```
1 mkdir -p $HOME/bin
2 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!



```
1 nextflow run rsuhecki/hello
```

Nextflow will pull the `rsuhecki/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/rsuhecki/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`.

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

Processes and channels

- *process* – a wrapper for a language-agnostic script which ensures isolation of the executed code.
- *channel* – an asynchronous¹ FIFO queue which facilitates data flow to/from/between processes by linking their outputs/inputs.

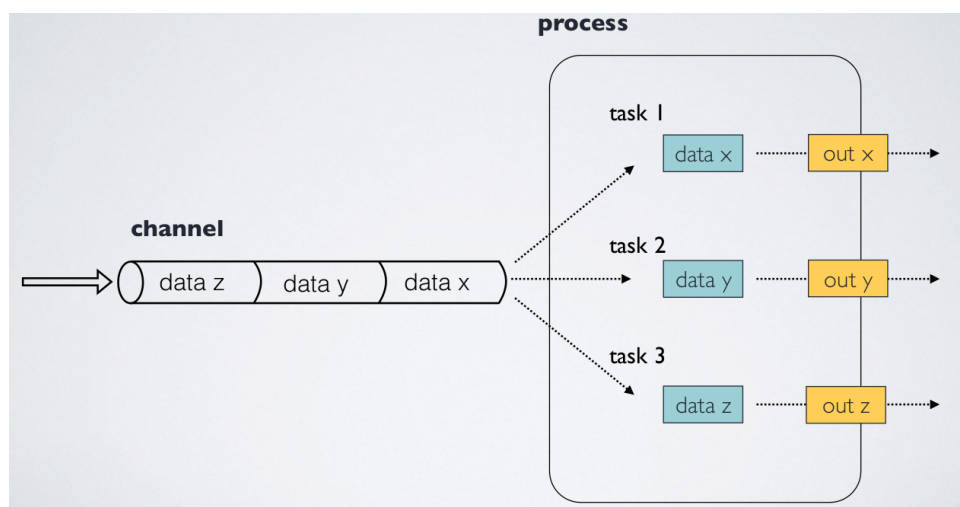


Figure 4: Nextflow building blocks: a *channel* “feeding” a processes. A *task* is an instance of a process. An isolated task is created for each emission (data chunk) from the input channel. Credit: Evan Floden

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 {

```

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

```

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 content 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 rsuckecki/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 rsuckecki/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 rsuckecki/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 rsuckecki/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.

```
1 mkdir -p /shared/${USER}/nextflow-tutorial
2 cd /shared/${USER}/nextflow-tutorial
3 git clone \
    https://github.com/csiro-crop-informatics/nextflow-embl-abr-webinar.git \
    example_workflow
4 cd example_workflow
5 git checkout noslurm
6 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.

```
1 nextflow run setup_data.nf
```

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

```
1 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?

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:

```
1 process foo {  
2   executor 'slurm'  
3   module 'samtools/1.9'  
4   //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.

```
1 #nextflow.config
2
3 process.executor = 'slurm'
4 process.module = 'samtools/1.9'
```

or using the preferred syntax

```
1 process {
2     executor = slurm
3     module = 'samtools/1.9'
4 }
```

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 or 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:

```
1 profiles {
2     //SOFTWARE
3     conda {
4         process {
5             conda = "$baseDir/conf/conda.yaml"
6         }
7     }
8     singularity {
```

```
9     process {
10         container = \
            'shub://csiro-crop-informatics/nextflow-embl-abr-webinar'
11     }
12     singularity {
13         enabled = true
14         autoMounts = true
15         cacheDir = "singularity-images"
16     }
17 }
18 }
```

As you can see, Nextflow makes it really easy to define software environment via Singularity or Conda².

Given that Singularity 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.



```
1 # Load the Singularity module
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
8 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 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.

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



What is your `slurm` profile configuration and where do you place it in `nextflow.config`?

There are of course many settings 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 defined⁵. 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
8 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>

⁵If 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.



```
1 # Remove the work directory to limit the number of task directories \
  to look at
2 rm -r work
3 # Re-run for a single sample
4 nextflow run main.nf -profile slurm,singularity
5 # Take a peak
6 ls -la work/ | less
7 # or
8 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

```
1 # NF generated files
2 .command.begin
3 .command.err
4 .command.log
5 .command.out
6 .command.run
7 .command.sh
8 .command.trace
9 .exitcode
10
11 # Output file
12 H45.bam
13
14 # Symlinks to input files
15 H45_R1.paired.fastq.gz
16 H45_R2.paired.fastq.gz
17 reference.fasta.gz.amb
18 reference.fasta.gz.ann
19 reference.fasta.gz.bwt
20 reference.fasta.gz.pac
21 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.



```
1 | 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](https://www.nextflow.io/docs/latest/operator.html) 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? Recall that workflow parameters use double-dash syntax. Run the relevant commands.

^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`:

```
1 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:

```
1 sq
2
```

```

3 | # Someone elses jobs
4 | sq --user ${SOMEONE_ELSE}

```

If you want to see all jobs in the queue:

```

1 | 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.

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

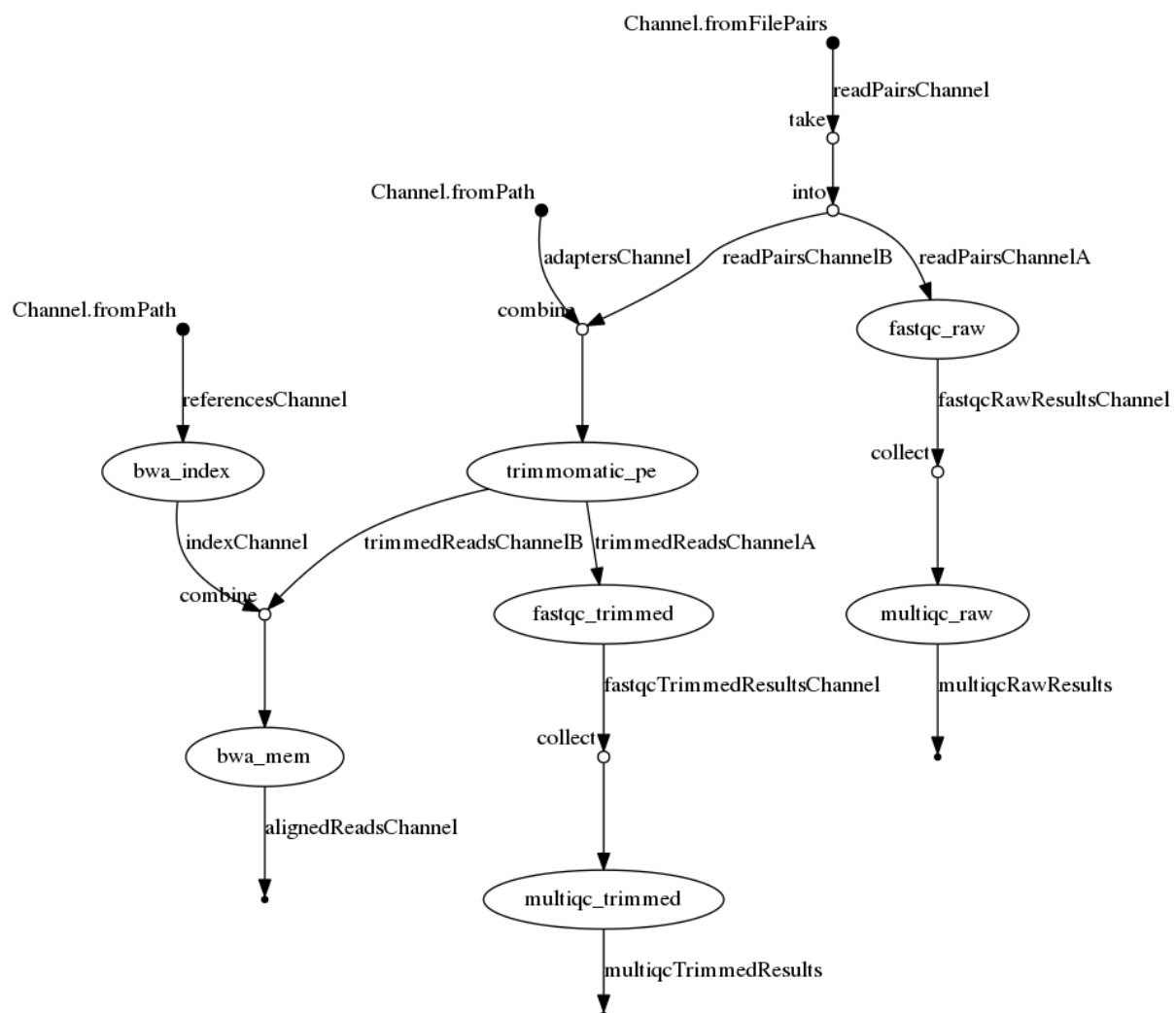


Figure 5: The example workflow



Investigate `main.nf` alongside Figure 5.

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

^a<https://www.nextflow.io/docs/latest/operator.html>

Workflow outputs

We now have each task 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 gathered 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.

⁶<https://www.nextflow.io/docs/latest/process.html#publishdir>

⁷<https://www.nextflow.io/docs/latest/config.html#process-selectors>



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

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



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.

Troubleshooting

Disconnected from the cluster?

Missing modules - new shell session?



Make sure all the required modules are loaded.

```
1 # Java - essential for nextflow
2 module load openjdk-1.8.0_202-b08-gcc-5.4.0-sypwasp
3
4 # Singularity - our go to system for providing software for the example \
   workflow
5 module load singularity-3.2.1-gcc-5.4.0-tn5ndnb
6
7
8 # If using conda
9 module load miniconda3-4.6.14-gcc-5.4.0-kkzv7zk
```

Space for Personal Notes or Feedback

This image shows a single sheet of white paper with horizontal ruling lines. The lines are evenly spaced and run across the width of the page. There is no text or other markings on the paper.

[illegible]

[illegible]

[illegible]