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:

```
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 convienently 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:

```
git checkout tutorial

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
```

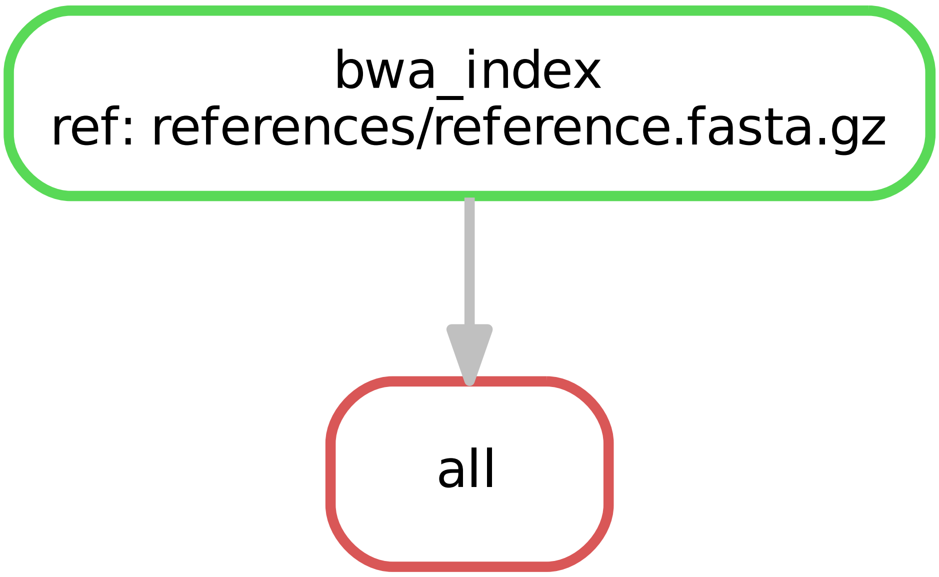

Figure 1: DAG of jobs showing `bwa_index` job dependant on the `all` pseudo-rule.

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` list:

```
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 list 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` list, 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
```

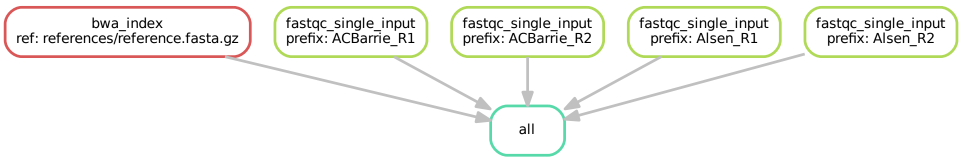

Figure 2: DAG of jobs showing `bwa_index` and several `fastqc_single_input` job dependant on the `all` pseudo-rule.

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
```

Depending on how quickly everyone else is in executing their workflows, you might get to see your jobs in the Slurm queue by 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!

If you compare the `trimmomatic` command in `analysis.sh` to the rule provided below, you will see that we have simply pulled out all references to input or output files into the `input` or `output` directives. Where we had used the bash variable `${SAMPLE}` in the filenames we are now using Snakemake "wildcards". It is almost the same syntax - just notice the absence of the $ but the curly braces are retained. The biggest changes seen are in the `shell` directive, where we now have to refer to the input and output files via `{input.r1}`, `{output.r1_unpaired}` etc.

```
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']),
```

Implementing BWA-MEM

Now is your opportunity to put into practice what you have learnt from the above walk-thoughs of implementing FastQC and Trimmomatic commands. Your task is to implement the `bwa mem` command into a Snakemake rule.

Here are some questions to get you thinking as you try to implement this rule:

What input read files are required for the command/rule?

QC'd R1 reads for a sample: `qc_reads/{SAMPLE}_R1.fastq.gz` QC'd R2 reads for a sample: `qc_reads/{SAMPLE}_R2.fastq.gz`

Does the command/rule need the FASTA reference file or the index files as input?

The rule doesn't need the FASTA, it needs the index files:

- `references/reference.fasta.gz.amb`
- `references/reference.fasta.gz.ann`
- `references/reference.fasta.gz.bwt`
- `references/reference.fasta.gz.pac`
- `references/reference.fasta.gz.sa`

The BWA-MEM command uses a "prefix" to the FASTA index files, not the index filenames themselves. How will you specify this in the `shell` directive?

If you hard-coded it, what would the rule look like?

```
shell:
  """
  bwa mem -t 1 \
    references/reference.fasta.gz \
    {input.r1} {input.r2} \
  | samtools view -b \
  > {output}
  """
```

Q

Hard-coding the path is simple, but not ideal. What if you changed the name of the reference file or wanted to use the rule with a different project? You would have to modify the paths in multiple places, once in the `input` directive and once in the `shell` directive.

Moving the hard-coded path out of the `shell` directive into the `params` directive (see: https://snakemake.readthedocs.io/en/stable/snakefiles/rules.html#non-file-parameters-for-rules).

```
params:
  prefix = "references/reference.fasta.gz",
shell:
  """
  bwa mem -t 1 \
    {params.prefix} \
    {input.r1} {input.r2} \
  | samtools view -b \
  > {output}
  """
```

Now you are making use of the `params` directive, this opens up the possibility of using some Python to do some string manipulations on the paths defined in the `input` directive. In particular, we can use a Python lambda function in the `params` directive (see: https://snakemake.readthedocs.io/en/stable/snakefiles/rules.html#non-file-parameters-for-rules).

How might you change a hard-coded path in the `params` directive to use a lambda function which manipulates the index file path(s) set in the `input` directive to define the prefix? Hint: take the path of one of the index files and remove the last few characters corresponding to the last file extension. You will probably need to do some reading of the Snakemake and/or Python documentation.

```
input:
  reference = expand("references/reference.fasta.gz.{ext}", \
      ext=["amb","ann","bwt","pac","sa"]),
...
params:
  prefix = lambda wildcards, input: input["reference"][0][:-4],
shell:
  """
  bwa mem -t 1 \
    {params.prefix} \
    {input.r1} {input.r2} \
  | samtools view -b \
  > {output}
  """
```

Now add the BAM files corresponding to the all the samples in the `SAMPLES` list, that you want the BWA-MEM rule to produce, to the `all` pseudo-rule.

```
input:
  reference = expand("references/reference.fasta.gz.{ext}", \
      ext=["amb","ann","bwt","pac","sa"]),
...
params:
  prefix = lambda wildcards, input: input["reference"][0][:-4],
shell:
  """
  bwa mem -t 1 \
    {params.prefix} \
    {input.r1} {input.r2} \
  | samtools view -b \
  > {output}
  """
```

Don't worry if you didn't complete the above implementation of the BWA-MEM command, we have a git repository branch with the rules we developed. Get it, execute the workflow and generate the DAG:

```
# Checkout the branch with our implemetation of these rules
git checkout final

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

# Generate a DAG
snakemake \
  --dag \
| dot -Tpdf \
> dag3.pdf
```

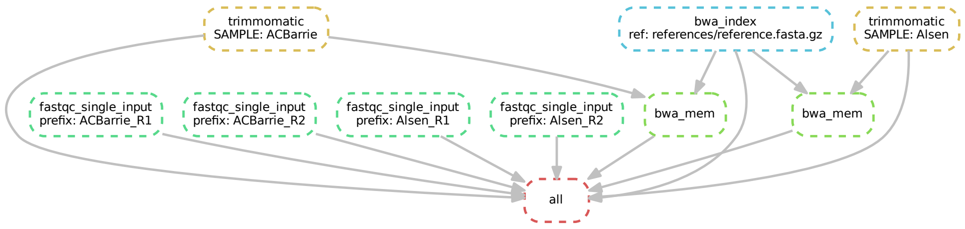

Figure 3: DAG of jobs showing the dependencies which exist in our final implementation of the `analysis.sh` workflow.

Adding New Samples

Our `SAMPLES` list contains a lot of samples which are currently commented out. Lets uncomment them and have a look at some other features of Snakemake:

```
# Manually uncomment the samples or use this sed command
sed -i 's/^# "/ "/' Snakefile
```

With so many more samples, the DAG becomes next to useless:

```
# Generate a DAG
snakemake \
  --dag \
| dot -Tpdf \
> dag4.pdf
```

Figure 4: DAG of jobs for the whole workflow consisting of 16 samples.

Instead, the “rulegraph” might provide a better view of the workflow. Unlike the DAG, that shows the individual jobs and their dependencies, the rulegraph shows only the rules and and their dependencies so provides a simplified view of the workflow:

```
# Generate a rulegraph
snakemake \
  --rulegraph \
| dot -Tpdf \
> rulegraph.pdf
```

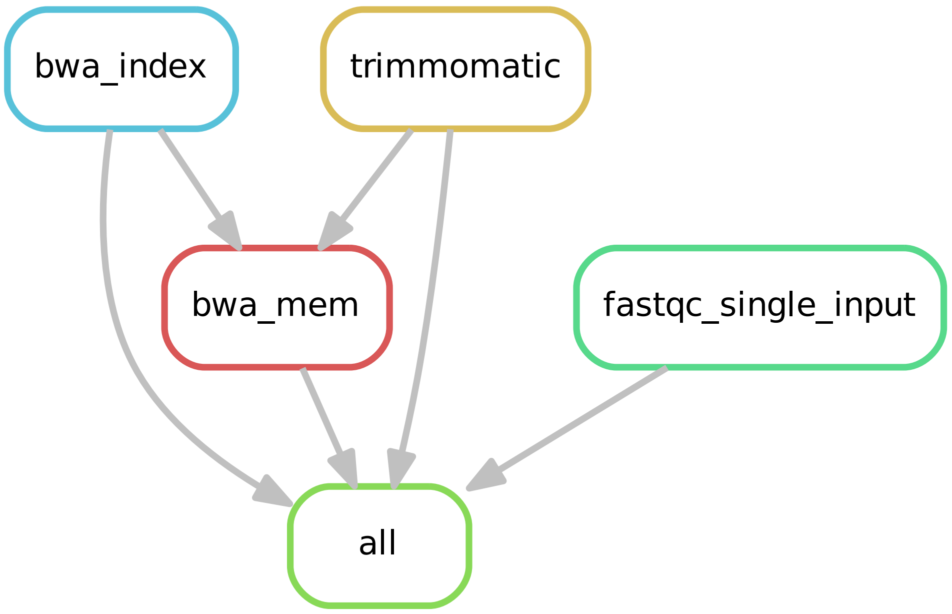

Figure 5: Rulegraph for the whole workflow.

Execute the rest of the workflow:

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

How many `fastqc_single_input` and total number of jobs are run as part of the whole workflow? Hint: try using `--forceall` in combination with `--dryrun`.

`fastqc_single_input`: 32

Total: 66

Using the Snakemake help, which command line argument can be used to get Snakemake to print the shell commands associated with each job during a `dryrun`?

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

Using the Snakemake help, which command line argument can be used to delete all the outputs associated with a given "target"?

```
snakemake \
  --delete-all-outputs
```

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 -version
```

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.

For the duration of the workshop we may fix the version of nextflow we will use, and also opt for simpler but more verbose terminal logging which may be more useful when you get started with Nextflow or when you start developing a new pipeline.

```
export NXF_VER=19.04.0
export NXF_ANSI_LOG=false

# now try
nextflow -version
```

To revert to the default logging you can either change the value of the environmental variable to `false` or use `-ansi-log false` at run time.

Running Nextflow

We start with Nextflow take on Hellow world.

Hello (nextflow) world!

Set-up a working directory for this task:

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

Clone the hello-world example and go to its directory

```
git clone https://github.com/rsuchecki/hello.git
cd hello
```

Feel free to investigate the content of the `main.nf` script, but we will have a closer look further on.

```
cat main.nf
```

Run the `main.nf` script and observe the terminal output

```
nextflow run main.nf
```

Run the hello script again a few times and observe the effect of using the `-resume` flag

```
nextflow run main.nf
nextflow run main.nf -resume
```

You may run the hello script a few more times with and without `-resume`.

How does the output differ depending on whether the `-resume` flag is used? With `-resume`, Nextflow will not re-run the tasks and report them as `cached`.

Hello command line options

Single-dashed options are reserved for Nextflow engine (`-resume, -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 the variable `params.universe` which is by default set to 'World', so lets try to use an alternative string.

```
nextflow run main.nf -resume --universe Mundo
```

What effect did the `-resume` flag had this time? The use of `-resume` was inconsequential as the process `sayHello` was never previously executed with input that includes the value of `params.universe` set to `'Mundo'` rather than `'World'`

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

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

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.

For more on task caching see https://www.nextflow.io/docs/latest/process.html#cache

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.

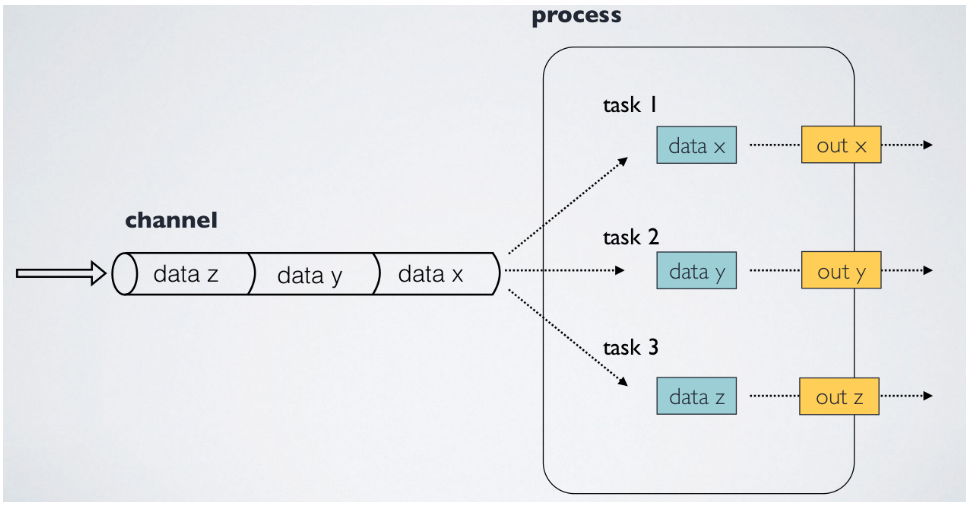

Figure 6: Nextflow building blocks: a *channel* "feeding" a processes. A *task* is an instance of a process. An isolated task is created for each emission (data chunk) from the input channel. *Credit: Evan Floden*

The 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:

```
#!/usr/bin/env nextflow

cheers = Channel.from 'Bonjour', 'Ciao', 'Hello', 'Hola', 'Czesc'

params.universe = 'World'

process sayHello {
  echo true

```

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

```
10    input:
11      val cheer from cheers
12
13    script:
14      """
15      echo "$cheer $params.universe!"
16      """
17  }
```

A channel called `cheers` is created and emits each of the listed strings separately. A separate instance of the process `sayHello`, i.e. a *task*, is executed for each emission.
The content of the above script can be broken down as follows:

- The shebang line (line 1) is optional.
- `Channel.from(some_list)` creates a channel emitting the list elements one by one.
- `params.universe = 'World'` sets the default value of the command-line accessible option (e.g. `--world Mundo`)
- Process definition (lines 7-16)
  - Directives are placed at the top, directive `echo` controls whether `stdout` of the process is redirected to your terminal
  - Input block (lines 10-11)
  - Script block (lines 13-16)
    - It is assumed to be a bash script unless an alternative shebang line (e.g. `/usr/bin/env python3`) is specified at the top of the block
    - The `$cheer` in the script block is a nextflow variable local to the process, **not** a bash variable.
- Indentation is inconsequential.

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.

Thanks to Nextflow's out-of-the-box support for different compute environments, it only requires changing the process executor from the default `local` to `slurm`. This could be done via configuration files (more about that later) or using the directive `executor 'slurm'` in the process definition. However, for our simple use-case it should suffice to use the `NXF_EXECUTOR` environmental variable.

```
NXF_EXECUTOR=slurm nextflow run main.nf
```

Play spot-the-difference – how can you tell if the tasks were submitted on the cluster? The terminal output should include the line `executor > slurm (5)` (or similar), but may differ depending on the version of Nextflow and whether ANSI logging is switched on or off.

Let's modify our script slightly to make it easier to see how processes are executed. Replace the single line within the script block

```
echo "$cheer $params.universe!"
```

with the following

```
echo "$cheer ${params.universe}!  -- from ${task.executor} \$HOSTNAME"
```

Note the difference between how nextflow variables (`$x`,`$params.universe`) 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. See for example Nextflow documentation of the alternative `shell` block[a].

[a] https://www.nextflow.io/docs/latest/process.html#process-shell

Now re-run...

```
NXF_EXECUTOR=slurm nextflow run main.nf
```

The messages printed to the terminal should by each task should now include the name of the executor being used and the name of the compute node on which the task was executed.

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.

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

Reimplementing a workflow in Nextflow

In the Snakemake section of our tutorial, we include an example bash "pipeline", which includes a number of common tasks wrapped in a single bash script[2]. The workflow includes the following steps

- Run FastQC across the raw read (FASTQ) 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

Before we start developing the Nextflow script, we need to set-up

- shared code base to be able to keep our efforts in sync when required
- input data for our workflow
- software environment including third-party tools

Getting the code

Start by cloning the workflow repository and moving to the directory.

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

[2]https://github.com/UofABioinformaticsHub/snakemake_template/blob/tutorial/analysis.sh

Getting the Data

We have provided you with some real whole genome sequencing (WGS) data from bread wheat together with a small chunk of the wheat genome. The data set is small enough for each step in the analysis to take less than a couple of minutes to run. We have a copy of this data available locally to save on bandwidth, 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/ \
  ./

# Have a look at what files we'd provided
tree data
```

To be able to run this tutorial outside the HPC cluster provided for the workshop you may use the provided script to download the required input files

```
#nextflow run setup_data.nf
```

The software environment

There are multiple ways in which you can provision a software environment for your workflow. You may have al the required tools installed on your system or available as a collection of pre-compiled binaries? Perhaps a friendly sysadmin installs convenient modules on your HPC cluster? You may also be using conda or perhaps docker or singularity containers? All those options are compatible with Nextflow.

We have provided a rudimentary script `versions.nf` which tries to run each of the software tools, and report their version numbers.

Try to run

```
nextflow run versions.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`).

Identify the exact `error`. The cause was likely "`.command.sh: line 2: bwa: command not found`". See line 23 in the following example:

```
 1  N E X T F L O W ~version 19.04.0
 2  Launching `versions.nf` [elegant_morse] - revision: 525c4d8fea
 3  [warm up] executor > local
 4  [0a/26fff6] Submitted process > get_versions
 5  ERROR ~Error executing process > 'get_versions'
 6
 7  Caused by:
 8    Process `get_versions` terminated with an error exit status (127)
 9
10  Command executed:
11
12    fastqc --version
13    multiqc --version
14    echo -n "bwa " && bwa 2>&1 | grep 'Version'
15    echo -n "samtools "&& samtools 2>&1 | grep 'Version'
16
17  Command exit status:
18    127
19
20  Command output:
21    (empty)
22
23  Command error:
24    .command.sh: line 2: fastqc: command not found
25
26  Work dir:
27    /shared/training001/nextflow/example_workflow/work/0a/26fff6b11f473b0c3f0b3961608462
28
29  Tip: view the complete command output by changing to the process \
        work dir and entering the command `cat .command.out`
30
31   -- Check '.nextflow.log' file for details
```

There are different ways in which we could ensure that the relevant software is available, for example using process *directives* at the top of each process definition. We could also use the opportunity to change the executor from local to Slurm, for example:

```
process foo {
  module 'samtools/1.9'
    executor 'slurm'
  //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 from `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

Have a look inside `nextflow.config`, and locate the process definitions

```
less nextflow.config
```

The ones most immediately relevant are:

```
profiles {
  //EXECUTORS
  slurm {
    process {
      executor = 'slurm'
    }
  }
  //SOFTWARE
  conda {
    process {
      conda = "$baseDir/conf/conda.yaml"
    }
  }
  singularity {
    process {
      container = '/shared/.singularity/nextflow-embl-abr-webinar.simg'
    }
    singularity {
      enabled = true
      autoMounts = true
    }
  }
}
```

As you can see, Nextflow makes it really easy to define software environment via Singularity or Conda[3].

Given that Singlularity is available on our cluster, we will use the `singularity` profile for software environment using `-profile singularity`. Combinations of profiles can be selected at runtime so we will also use the `slurm` profile which will ensure the tasks are not executed on the head node but submitted to the cluster.

In addition to this basic configuration, there are many setting that can and in some cases must be set – refer to executors section of Nextflow documentation[4]. For running real-life pipelines in a cluster environment you will also use directives[5] controlling the resources

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

[4]`https://www.nextflow.io/docs/latest/executor.html`

[5]`https://www.nextflow.io/docs/latest/process.html#directives`

(`cpus, memory, time`) requested for each job. Other directives relevant in HPC context might include `queue` and `scratch`.

```
# 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 versions.nf -profile slurm,singularity
```

Reimplementing a workflow in Nextflow - a walk-through

Recall the steps of the original bash workflow

- Run FastQC across the raw read (FASTQ) 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 will walk you through reimplementing the first few steps into a Nextflow workflow. Along the way, we will introduce the core concepts of Nextflow and then ask you to reimplement the `bwa-mem` step yourselves. If you finish that with some time to spare, you will have the opportunity to reimplement the `multiqc` step. This will provide you with a foundation for you to be able to port your own workflows into Nextflow.

Implementing FastQC

In this section you will learn how to

- Read multiple input files into a *channel*
- View items emitted by a *channel*
- Restrict how many items are emitted through a *channel* using *operators*
- Effortlessly parallelize the task by plugging a *channel* into a *process*

- Reference emitted items within processes' script block

Paste and save the following into a new file `main.nf`

```
Channel.fromPath("data/raw_reads/*.fastq.gz")
  .view()
```

We create a (yet unnamed) channel and use the `view` *operator* to print the names of emitted files to the terminal. If you now run

```
nextflow run main.nf
```

You will see a long list of files being emitted by the channel.

While developing a workflow we tend to prefer to only use a small subset of data. We could restrict the number of FASTQ files emitted by the channel e.g. by pointing to a specific file or a subset of files, in this case the two read files available for Baxter

```
Channel.fromPath("data/raw_reads/Baxter_R{1,2}.fastq.gz")
  .view()
```

We can do better! Let's use one of NF operators capable of limiting the number of items emitted by the channel. Either **one** of `first()`, `last()` or `take(n)` will suffice.

```
Channel.fromPath("data/raw_reads/*.fastq.gz")
  .take(1)
  .first()
  .last()
  .view()
```

Having all three operators is redundant, we will stick with `take(1)`, due to extra flexibility it gives us. The last thing to do is to assign our channel to a variable whose name we can use elsewhere in the script. The simplest form would be

```
readsForQcChannel = Channel.fromPath("data/raw_reads/*.fastq.gz")
  .take(1)
  .view()
```

But since we often chain multiple operations, the following syntax is preferred due to consistent left-to-right flow.

```
Channel.fromPath("data/raw_reads/*.fastq.gz")
  .take(1)
  .view()
  .set { readsForQcChannel }
```

Update `main.nf` and run it again

```
nextflow run main.nf
```

You should see that only a single item is emitted by the channel.

Now that we have our input channel ready and emitting, we can remove the view operator and focus on defining a process to implement the original bash snippet.

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

In this case, there is no benefit from processing files as pairs. In fact, separating the pairs should in principle allow more efficient allocation of resources as we will end up with twice the number of tasks, but each being half the size of the original.

In `main.nf` we start our process definition by giving it a name and defining the input block

```
process fastqc {

  input:
   file(reads) from readsForQcChannel

}
```

We defined process input to be file taken from the `readsForQcChannel` which we assign to variable `reads`. We can now proceed to defining our script block.

```
process fastqc {

  input:
   file(reads) from readsForQcChannel

  script:
  """
  fastqc --threads 1 ${reads}
  """
}
```

In the script block we address the input file via its variable name as `$reads` or `${reads}`.

Rather than fixing the number of threads inside the script block to 1, we could use the `task.cpus` variable.

How can you set the value of the `task.cpus` variable? Hint: look into *process directives* documentation.

```
process fastqc {
  cpus 2

  input:
   file(reads) from readsForQcChannel

  script:
  """
  fastqc --threads ${task.cpus} ${reads}
  """
}
```

Implementing BWA Indexing

In this section you will learn how to

- group *process* output files and values in a `set` to be emitted through an output channel

In the original bash script this task consists of a simple call to `bwa` with the input fie name hard-coded.

```
bwa index -a bwtsw references/reference.fasta.gz
```

We start by creating a channel which pass the reference file to our indexing process.

```
referencesChannel = \
    Channel.fromPath('data/references/reference.fasta.gz')

# or the equivalent

Channel.fromPath('data/references/reference.fasta.gz').set { \
    referencesChannel }
```

We can now move on to defining the process which will *consume* the `referencesChannel`.

The bare-minimum would probably be a process definition with * an `input` block * a `script` block

```
process bwa_index {
  input:
    file(ref) from referencesChannel

  script:
```

```
  """
  bwa index -a bwtsw ${ref}
  """
}
```

That should be sufficient to run the indexing, but given that we want to use the generated index, in another process, we also need to define the outputs for the current one. The expected output files are:

```
reference.fasta.gz.amb
reference.fasta.gz.ann
reference.fasta.gz.bwt
reference.fasta.gz.pac
reference.fasta.gz.sa
```

To capture these as outputs we could simply declare that all (non-input) files constitute the desired output.

```
  output:
    file("*") into indexChannel
```

We could also be slightly more explicit and declare that outputs are all (non-input) files sharing a prefix

```
  output:
    file("${ref}.*") into indexChannel
```

or even painstakingly list all the expected files... not recommended. However, one of the great things about Nextflow is that thanks to process isolation, you typically don't have to pay much attention to input/output file names unless a tool you use have some specific requirements in this regard. For example, to run `bwa` alignment later on, we will need to pass it the common prefix of the names of the index files.

In this case rather then simply outputting the index files, and then parsing the prefix out of the file name(s), we opt to output our reference file name alongside the generated index file. To do that in a single emission we wrap these in a *set* (in other words a tuple - an ordered list of elements).

```
  output:
    set val("${ref}"), file("*") into indexChannel
```

The emitted set will consist of two elements

- The reference file name (String)
- The list of index files (not just file names – objects of class `java.nio.file.Path`).

This is our complete indexing process definition

```
process bwa_index {

  input:
    file(ref) from referencesChannel

  output:
    set val("${ref}"), file("*") into indexChannel

  script:
  """
  bwa index -a bwtsw ${ref}
  """
}
```

Give it a whirl,

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

Implementing Trimmomatic

We will guide you through the implementation of one more process before you take over! The task is to adapter/quality trim raw reads as pairs and output them into a channel to be consumed by an alignment process.

The original bash implementation was as follows

```
  #####
  mkdir -p qc_reads
  trimmomatic PE \
      -threads 1 \
      raw_reads/${SAMPLE}_R1.fastq.gz raw_reads/${SAMPLE}_R2.fastq.gz \
      qc_reads/${SAMPLE}_R1.fastq.gz \
         qc_reads/${SAMPLE}_R1.unpaired.fastq.gz \
      qc_reads/${SAMPLE}_R2.fastq.gz \
         qc_reads/${SAMPLE}_R2.unpaired.fastq.gz \
      ILLUMINACLIP:misc/trimmomatic_adapters/TruSeq3-PE.fa:2:30:10:3:true \
         \
      LEADING:2 \
      TRAILING:2 \
      SLIDINGWINDOW:4:15 \
      MINLEN:36
```

This time we have to process the reads **as pairs** so using `Channel.fromPath()` will not suffice. Conveniently, NF has been developed by bioinformaticians so it smoothly handles paired files.

```
Channel.fromFilePairs("data/raw_reads/*_R{1,2}.fastq.gz")
  .take (1)
  .view ()
  .into { readPairsForTrimmingChannel }
```

As before, during development we

- use the `.take(n)` operator to limit the number of emissions coming through the channel
- use `.view()` to sneak-peak at what exactly is emitted

The output from the `.view()` operator in this case, after some re-formatting would be something like

```
[
  Gladius,
  [
    /path/to/data/raw_reads/Gladius_R1.fastq.gz,
    /path/to/data/raw_reads/Gladius_R2.fastq.gz
  ]
]
```

The single emission consists of a set of two elements.

1. The sample name (accession) i.e. the file names prefix captured by the '`*`' glob.
2. The list of two FASTQ files

One more thing we need, is a file containing the adapter sequences for trimmomatic. This could be done in a number of ways, but it is best to stick to the dataflow paradigm and set-up another channel for that.

```
Channel.fromPath('data/misc/trimmomatic_adapters/TruSeq3-PE.fa')
  .set{ adaptersChannel }
```

We now have two input channels, but how do we handle multiple input channels?

If we just declare each separately

```
process trimmomatic {
  input:
    set val(accession), file(reads) from readPairsForTrimmingChannel
    file(adapters) from adaptersChannel
...
```

This would work perfectly fine... but only for the first pair of reads. There is only a single adapters file emitted by the `adaptersChannel`, so once this channel no longer emits outputs, the process will not run again and will never read the next pair of FASTQ files from `readPairsForTrimmingChannel`.

What we really want is for the adapters file to be used in combination with each pair of FASTQ files. Enter the `combine()` operator.

```
process trimmomatic_pe {

  input:
    set file(adapters), val(accession), file(reads)
      from adaptersChannel.combine(readPairsForTrimmingChannel)
...
```

This covers our inputs and we can now focus on the outputs and the script block. As mentioned earlier we don't have to be too specific about the output file names but just have to make it clear to NF which output files are to be sent to an output channel. In this case these are the ones with names ending with `paired.fastq.gz`. We also output the `accession` variable to keep track of which sample is being processed.

```
process trimmomatic_pe {

  input:
    set file(adapters), val(accession), file(reads)
      from adaptersChannel.combine(readPairsForTrimmingChannel)
  output:
    set val(accession), file('*.paired.fastq.gz') into \
        trimmedReadsChannel
...
```

The translation of the trimmomatic call from plain bash to bash embedded in the script block is straightforward.

1. Explicit paths to the input files are replaced with `${reads}` variable. The read files could also be addressed individually as `${reads[0]}` and `${reads[1]}`.
2. Explicit output file paths for trimmomatic are replaced with convenient short names which need not be globally unique.

```
process trimmomatic_pe {

  input:
    set file(adapters), val(accession), file(reads)
      from adaptersChannel.combine(readPairsForTrimmingChannel)

  input:
    set file(adapters), val(accession), file(reads) from \
        adaptersChannel.combine(readPairsForTrimmingChannel)

  output:
    set val(accession), file('*.paired.fastq.gz') into \
        trimmedReadsChannel

  script:
  """
  trimmomatic PE \
    ${reads} \
    R1.paired.fastq.gz \
    R1.unpaired.fastq.gz \
    R2.paired.fastq.gz \
    R2.unpaired.fastq.gz \
    ILLUMINACLIP:${adapters}:2:30:10:3:true \
    LEADING:2 \
    TRAILING:2 \
    SLIDINGWINDOW:4:15 \
    MINLEN:36 \
    -Xms256m \
    -Xmx256m
  """
}
```

Implementing BWA-MEM

Now is your opportunity to put into practice what you have learnt from the above walk-through of implementing FastQC and Trimmomatic commands. Your task is to implement the `bwa mem` command in a Nextflow process.

The relevant snippet from the bash pipeline is as follows

```
  bwa mem -t 1 \
    references/reference.fasta.gz \
    qc_reads/${SAMPLE}_R1.fastq.gz qc_reads/${SAMPLE}_R2.fastq.gz \
  | samtools view -b \
  > mapped/${SAMPLE}.bam
```

Here are some questions to get you thinking

The mapping process requires and index comprised of several files but takes their common prefix (basename) as argument. How can you pass that prefix to the command?
You *could* hard-code it (not a good practice) or parse it out from the index file names.
Can you do it without hard-coding it or having to parse it out from the input file name(s)?
Our indexing process, in addition to the index files also outputs the prefix used when building the index.
How do you make sure that each instance of the process - and not just the first one - gets the index files in *(cough)* combination with a pair of FASTQ files?
Use the *combine* operator to combine the `indexChannel` with the `trimmedReadsChannel`.

```
process bwa_mem {
  input:

  output:

  script:
  """
  bwa mem -t ${task.cpus} ${ref} ${reads} \
  | samtools view -b > ${accession}.bam
  """
}
```

Adding New Samples

Since we had to stop the channel from sucking up all the samples by applying the take operator we expect you may already know how to make sure all samples are processed.

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.

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.

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.

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.

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

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
```

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.

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.

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