THE UNIVERSITY OF UTAH

The Snakemake Workflow Manager

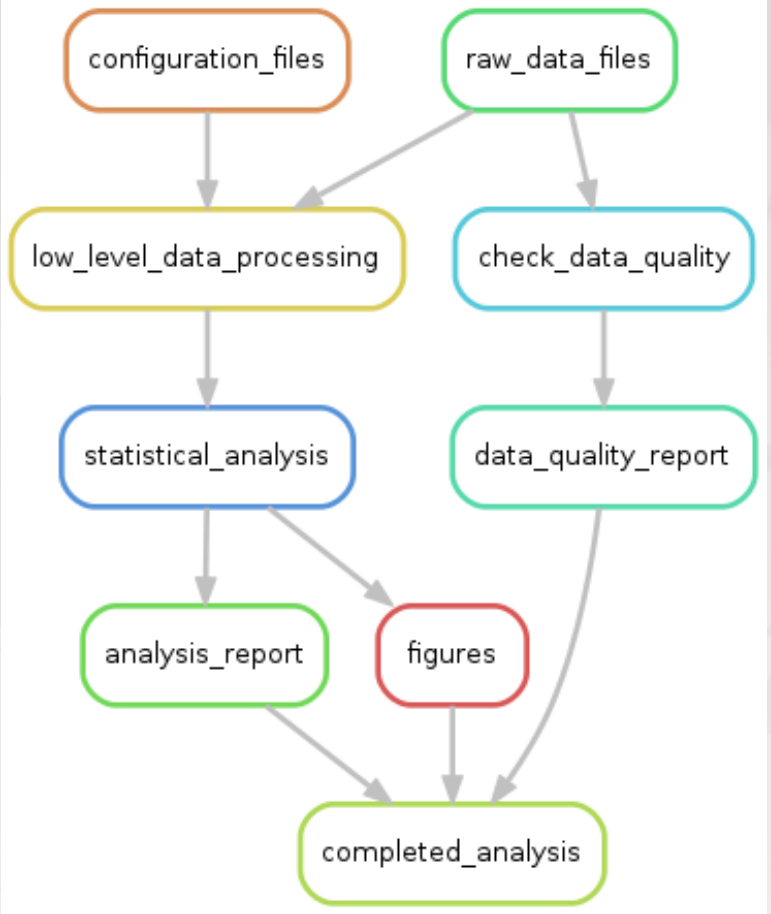
Brett Milash

Center for High Performance Computing

University of Utah

A workflow manager is software that:

- Conducts a complex work flow or analysis
- Follows dependencies from results back to source code / data files
- Executes statements step-by-step to carry out work flow



https://snakemake.readthedocs.io/en/stable/

Why use a workflow manager?

- Human efficiency: re-use and standardization
- Convenience
- Computational efficiency – only the required steps are executed
- Reproducibility
- Portability between clusters, institutions
- Modularity – divide and conquer!

Why choose snakemake?

Over 100 different workflow managers:
https://github.com/pditommaso/awesome-pipeline

Snakemake is:

- ❑Actively used and developed
- ❑Can be configured for local and/or cluster execution
- ❑Native SLURM support
- ❑No significant system administration support required
- ❑General purpose (i.e. not just for bioinformatics)
- ❑Significant functionality bang for your learning buck

Installation

- Use the CHPC module:
  - `module load snakemake/5.4.2`
- Install your own using pip:
  - `pip install --user snakemake`
  - `export PATH=$HOME/.local/bin:$PATH`
- Install your own using anaconda:
  - `module load anaconda3`
  - `conda install –c bioconda –c conda-forge snakemake`

Snakemake is a better make

Classical makefile:

Rule

Dependencies (input)

```
hello_world: hello_world.o
        gcc -o hello_world hello_world.o

hello_world.o: hello_world.c hello_world.h
        gcc -c hello_world.c
```

Target (output)

Action

Snakemake workflows are built out of rules

Rules can have:

- names
- inputs
- outputs
- actions (shell or python)

Rules:

- are linked implicitly
- (or explicitly)
- can emit messages
- are executed in parallel if possible
- are executed locally or on a cluster

The first rule defines the default “target” for the workflow

```
rule link:
        input: "hello_world.o"
        output: "hello_world”
        message: “Rule {rule} linking .o file {input}"
        shell: "gcc -o {output} {input}”
```

Snakefile syntax

- Snakemake work flows (”snakefiles”) are python code
- All the python syntax rules apply:
  - Input and output file names in quotes
  - Shell commands in quotes
  - Whitespace / indentation is significant
  - Use either tabs or spaces (not both)
- Your snakefile can include blocks of python code

Rule inputs

- Inputs are optional
- Inputs are one or more file names, in quotes, comma-separated
- Inputs can have “symbolic” names

```
rule align:
      input: fastq=“sample1.fastq”, index=“hg19”
      output: ”sample1.sam”
      shell: “bwa mem {input.index} {input.fastq} –o {output}”
```

Rule outputs

- Same as inputs: one or more file names, in quotes, comma-separated
- Same as inputs: can have ”symbolic names”
- Outputs are optional - common in top-level rule that simply checks if inputs are present.

Directories as input or output

- In snakemake >= 5.0:
  - Directories as input or output must be specified with directory()
    - input: directory(“data_directory”), “data_file”
- In snakemake < 5.0:
  - Directories as input or output are just named like regular files
    - input: “data_directory”, “data_file”

Rule messages

- Rules can emit messages with the “message:” section
- Messages are optional
- Really useful for monitoring your workflow
- Can access the inputs, outputs with {input}, {output}
- Can also access the rule name as {rule}

Rule actions: the “shell:” section

- This is where you encode the actual work of the work flow
- By default: /bin/bash in strict mode (set –euo pipefail)
- Multi-line shell statements: use triple-quotes
- Can load modules, only affects the current rule.

```
rule link:
        input: "hello_world.o"
        output: "hello_world"
        shell: """
                module load gcc/6.1.0
                gcc -o {output} {input}
                """
```

Rule “run:” section: action as python code

- Alternatively, implement action of the rule in python code
- Put this in the “run:” section of the rule

```
rule usercount:
        input: "userfile.txt"
        output: "users.count"
        run:
                users=set()
                with open(input[0]) as infile:
                        for line in infile:
                                unid=line.split()[0]
                                users.add(unid)
                with open(output[0],'w') as outfile:
                        print(f"There are {len(users)} users.",file=outfile)
```

Snakemake command line arguments

First, need to load the module:

```
module load snakemake/5.4.2
```

Run snakemake on default ”Snakefile”, default target:

```
snakemake
```

Run snakemake on non-default snakefile:

```
snakemake –s snakefile
```

Run snakemake on non-default target:

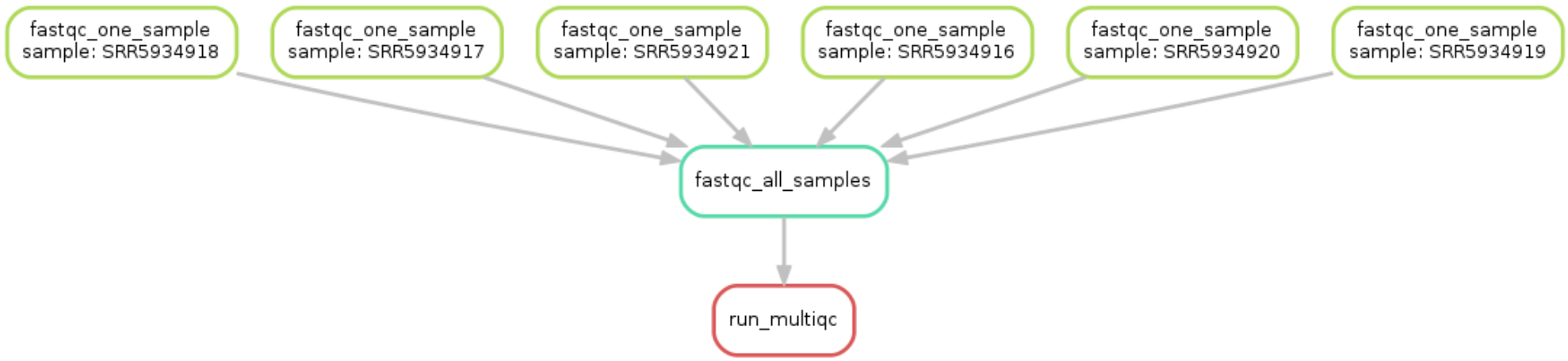
```
snakemake target_name
```

Read the snakemake help:

```
snakemake --help
```

Graphical output

- Rule graph
  - Shows in general how rules depend on one another, but not the actual inputs/outputs
  - `snakemake –s snakefile --rulegraph | dot –Tpng > rulegraph.png`
- Directed Acyclic Graph (DAG)
  - all targets represented
  - Completed rules have dashed outline
  - `snakemake –s snakefile --dag | dot –Tpng > dag.png`



Exercise 1 - Simple linear workflow

See the exercise 2 instructions here:

https://gitlab.chpc.utah.edu/bmilash/workflows-with-snakemake/tree/master/Exercises/Exercise1

Wildcards

Wildcards: filename patterns enable generalizable rules

- These make workflows reusable
- Parallelization!

How to do it:

- Create a rule that handles a single input -> output action using {variable} as a placeholder for the file name(s). This acts as a **template**.
- Create another rule whose **input** lists all the template rule's output files using the expand() function. Python lists and list comprehension are useful here.

Snakemake wildcard example

```
# Calculate the MD5 checksum for every .fastq.gz file.
# Here are the sample names embedded in the file names:
samples=[ 'SRR5934916',
        'SRR5934917',
        'SRR5934918',
        'SRR5934919',
        'SRR5934920',
        'SRR5934921' ]

rule all_checksums:
        input: expand("{sample}.md5", sample=samples)
        output: touch("checksums.done")    # Creates empty output file checksums.done

rule one_checksum:
        input: "{sample}.fastq.gz"
        output: checksum="{sample}.md5"
        shell: "md5sum {input} > {output.checksum}”
```

https://snakemake.readthedocs.io/en/stable/

Snakemake exercise 2

See the exercise 2 instructions here:

https://gitlab.chpc.utah.edu/bmilash/workflows-with-snakemake/tree/master/Exercises/Exercise2

Snakemake on a cluster

- Any snakemake task can run on a cluster:

```
snakemake -s snakefile --cluster-config cluster.yaml --jobs 20 …
```

- Cluster configuration file can be in JSON or YAML format
- The catch is we must tell snakemake how to start a job:
  - --cluster “sbatch –A {cluster.account} –p {cluster.partition}”

Cluster configuration

- Cluster configuration file:

```
# cluster.yaml - cluster configuration for my snakemake job.
__default__:
    partition: slurm_partition
    account: slurm_account
    time: 1:00:00
    nodes: 1
```

- Can provide default and rule-specific configurations

```
image_processing:
    partition: kingspeak_gpu
    account: kingspeak_gpu
```

Local rules

- When running on a cluster, may want to specify some rules NOT run on the cluster
- localrules: rule1, rule2, rule3
- Snakemake knows to run rules without a “shell:” locally.

Watching your workflow run on the cluster

- Run the squeue command to see your SLURM jobs:
  - squeue -i 3 -M all -u $USER
- You can get fancy with the output:
  - squeue -M all -i 3 -u $USER -o "%.6i %.10P %.7a %.20j %.2t %.6M %R"

Snakemake exercise 3

- See the exercise 3 instructions here:

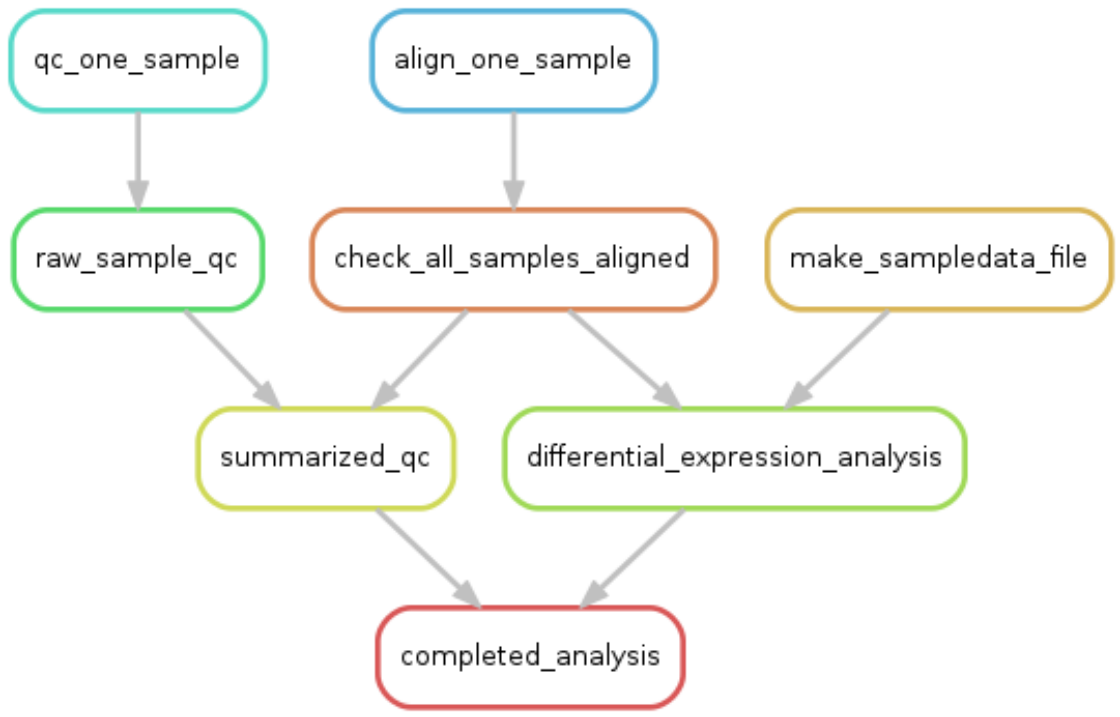
https://gitlab.chpc.utah.edu/bmilash/workflows-with-snakemake/tree/master/Exercises/Exercise3

- Create a cluster configuration file cluster.yaml:

```
# cluster.yaml - cluster configuration for my snakemake job.
__default__:
    partition: lonepeak
    account: chpc
    time: 1:00:00
    nodes: 1
```

- Run your snakefile (or use solutions/exercise2-2.snakefile) using your cluster configuration file.

Modular workflows




```
# Snakefile.qc
rule summarized_qc:
        input: …
        output: touch("qc.done")
        shell: …

rule qc_one_sample:
        input: …
        output: …
        shell: …

rule raw_sample_qc:
        input: …
        output: …
```

```
# In main Snakefile:
include: "Snakefile.qc"
```

Developing complex workflows

1. Define “skeleton” of workflow, linking rules together using touch().
2. Start at beginning, implementing one rule at a time, testing as you go.
3. Use a small data set for testing, fast feedback
4. Implement the cluster configuration
5. Re-test
6. Run it with real data set

Granularity

- Fine-grained
  - Many rules, simple shell statements
  - Efficient for local rules, easy debugging
  - Inefficient for cluster jobs, as each rule requires submitting a job
- Coarse-grained
  - Few rules, complex shell statements
  - More efficient on clusters

Snakemake is container-friendly

- Snakemake supports running code in containers using singularity
- See: https://snakemake.readthedocs.io/...#running-jobs-in-containers

Snakemake may not be right for you

- What if your inputs and outputs aren't files?
- What if your cluster doesn't use SLURM or LSF?
  - HTCondor (Open Science Grid: > 1.2 billion core hours last year)
- What if your workflow changes?
- nextflow: https://www.nextflow.io/
  - non-file inputs and outputs
  - support for HTCondor (OSG) and many other schedulers
  - workflow file is part of the workflow – when a rule changes, it gets re-run