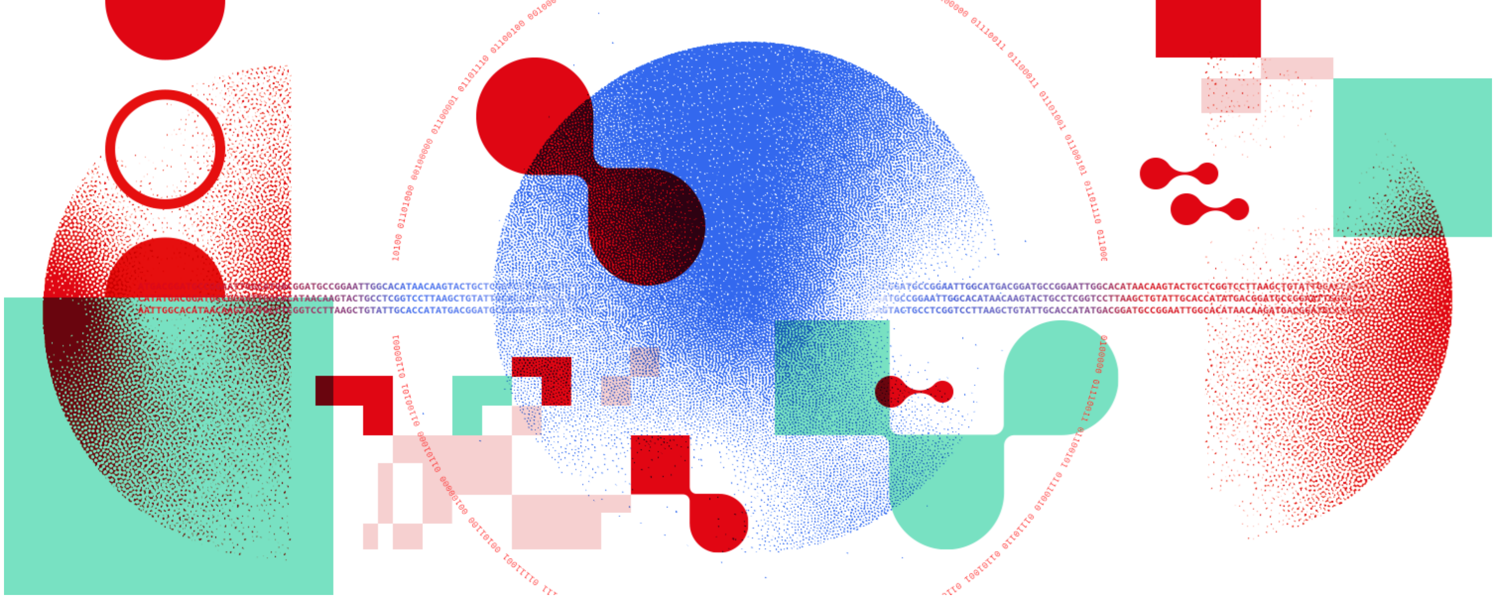


Snakemake for reproducible research

Making a more general-purpose Snakemake workflow

UNIL | Université de Lausanne

Antonin Thiébaut
antonin.thiebaut@unil.ch

SIB
Swiss Institute of Bioinformatics

Pop quiz

```
rule second_step:
    input:
        rules.first_step.output
    output:
        'results/first_step.txt'
    shell:
        'cp {input} {output}'
```

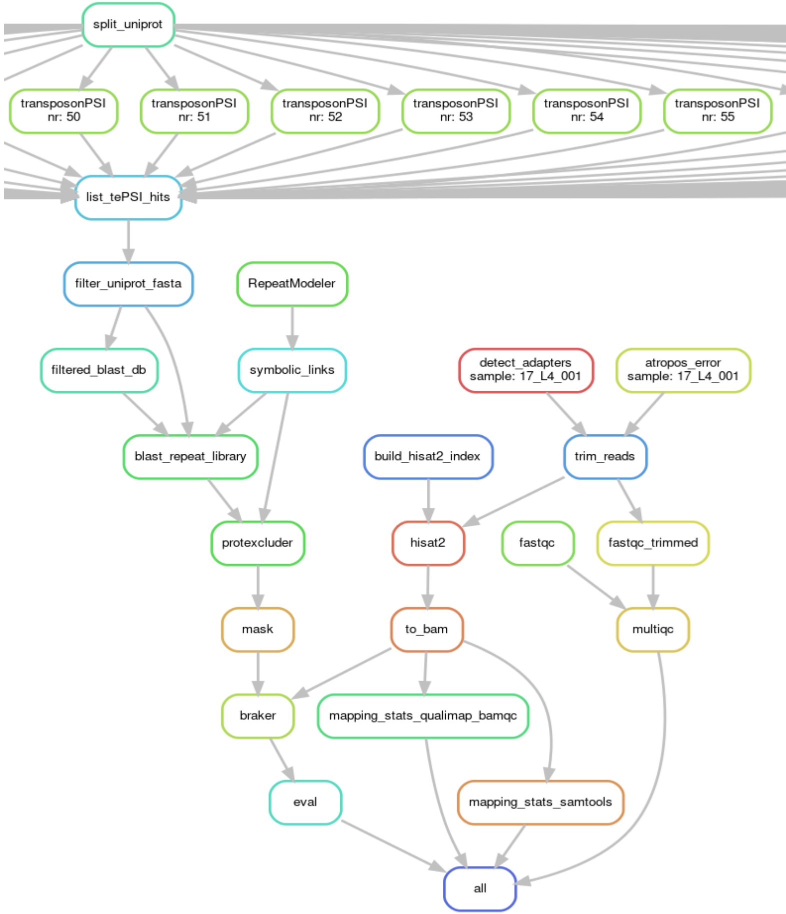
Pop quiz

- Snakemake keyword
- Rule name (user-defined)
- Snakemake directives
- Directives values:
  - Object
  - String (file path)
  - Instruction (command)
  - Numeric values (seen later)
- Mystery syntax?

```
rule second_step:
    input:
        rules.first_step.output
    output:
        'results/first_step.txt'
    shell:
        'cp {input} {output}'
```

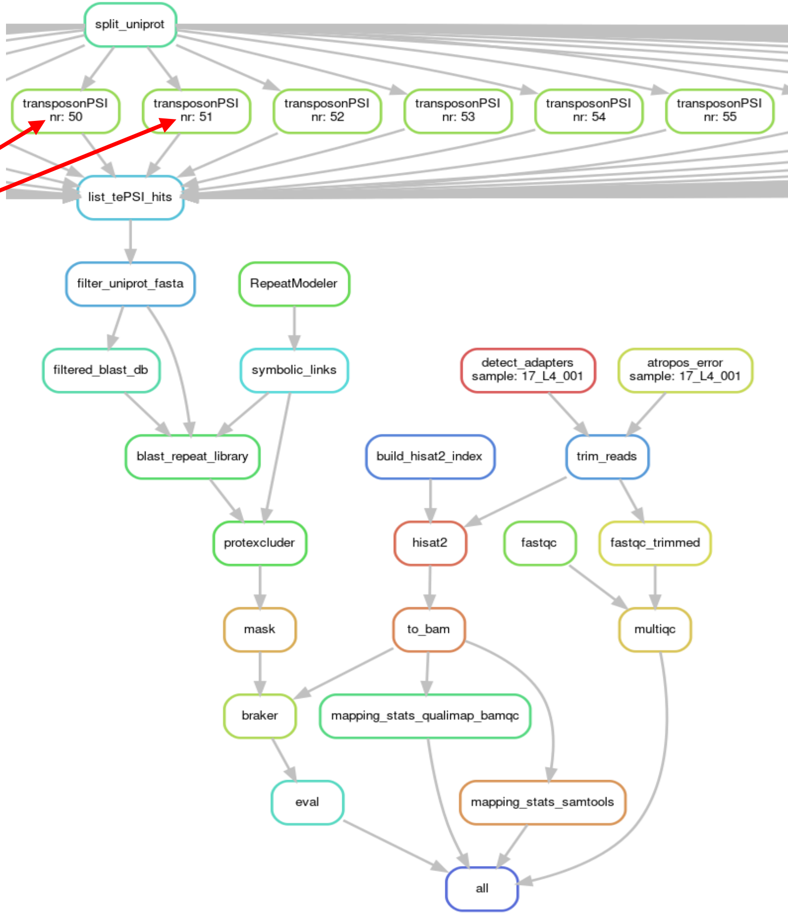
Building a Directed Acyclic Graph (DAG)

- Snakemake determines which jobs to run to produce desired outputs
  - DAG depends on Snakefile, requested target outputs, and files already present



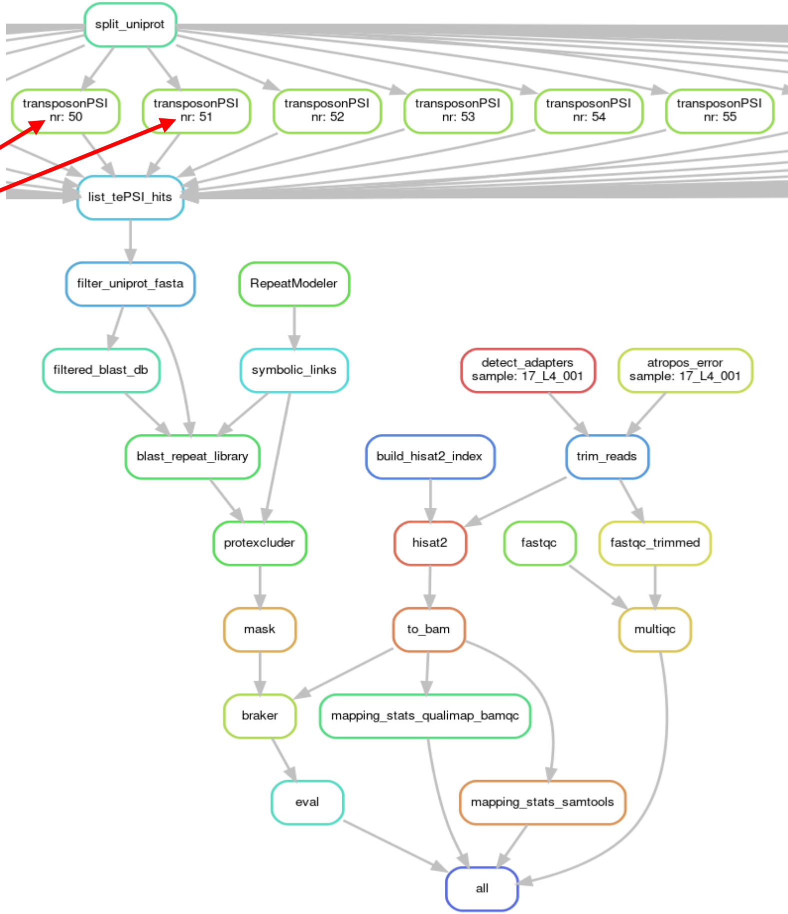
Building a Directed Acyclic Graph (DAG)

- Snakemake determines which jobs to run to produce desired outputs
    - DAG depends on Snakefile, requested target outputs, and files already present
- Rule can appear more than once, with different wildcards
    - 1 rule + 1 wildcard values = 1 job
- Arrows = dependency between jobs
    - Snakemake runs jobs in any order that doesn't break dependency



Building a Directed Acyclic Graph (DAG)

- Snakemake determines which jobs to run to produce desired outputs
    - DAG depends on Snakefile, requested target outputs, and files already present
- Rule can appear more than once, with different wildcards
    - 1 rule + 1 wildcard values = 1 job
- Arrows = dependency between jobs
    - Snakemake runs jobs in any order that doesn't break dependency
- DAG = work list, ≠ flowchart
    - No if/else decisions or loops
    - Snakemake runs every job in the DAG exactly once
- DAG does not check shell directives
    - Shell commands are tested during execution (1. Works? 2. Produces expected outputs?)



DAG (re-)run policy

- Snakemake runs a job if:
    - Target file explicitly requested is missing
    - Intermediate file is missing and needed to create target file
    - Input file is newer than an output file (timestamps comparison)
    - ➢ Can skip parts of the DAG

- Allows to:
    - Change/add inputs to existing analysis without re-running everything
    - Resume running a workflow that failed part-way

DAG (re-)run policy

- Snakemake runs a job if:
    - Target file explicitly requested is missing
    - Intermediate file is missing and needed to create target file
    - Input file is newer than an output file (timestamps comparison… but not only)
    - ➢ Can skip parts of the DAG

- Allows to:
    - Change/add inputs to existing analysis without re-running everything
    - Resume running a workflow that failed part-way

- Altering DAG (re-)run policy:
    - -f, --force <target_name>
    - -F, --forceall
    - -R, --forcerun <rule_name>
    - --rerun-triggers {mtime,params,input,software-env,code}
    - --touch

Several problems…

Several problems…

- Hard-coded file paths
- Processing list of files
- Only one input/output per rule
- Resources are not optimised

… that can be solved!

- Hard-coded file paths → Placeholders and wildcards
- Processing list of files → expand() syntax
- Only one input/output per rule → Numbered/named inputs/outputs
- Resources are not optimised → log, benchmarks, threads, memory…

Avoiding hard-coded filepaths: placeholders

- Placeholder:
    - A person or thing that occupies the position or place of another person or thing
    - A symbol in a mathematical or logical expression that may be replaced by the name of any element of a set

From the Merriam-Webster dictionary

Avoiding hard-coded filepaths: placeholders

```
rule example:
    input:
        'data/first_step.tsv'
    output:
        'results/first_step.txt'
    shell:
        'cp data/first_step.tsv results/first_step.txt'
```

Avoiding hard-coded filepaths: placeholders

```
rule example:
    input:
        'data/first_step.tsv'
    output:
        'results/first_step.txt'
    shell:
        'cp data/first_step.tsv results/first_step.txt'
```

```
rule example:
    input:
        'data/first_step.tsv'
    output:
        'results/first_step.txt'
    shell:
        'cp {input} {output}'
```

Avoiding hard-coded filepaths: placeholders

- {input} and {output} are placeholders
- Used in shell directive
- Similar to python f-string
- Snakemake will replace them with appropriate values before running the command
- Many directives can used in placeholders: {log}, {benchmark}, {params}…

```
rule example:
    input:
        'data/first_step.tsv'
    output:
        'results/first_step.txt'
    shell:
        'cp {input} {output}'
```

Making more general-purpose rules: wildcards

- Wildcards ≈ Snakemake "variables"

Making more general-purpose rules: wildcards

- Wildcards ≈ Snakemake "variables"

```
rule example:
    input:
        'data/first_step.tsv'
    output:
        'results/first_step.txt'
    shell:
        'cp {input} {output}'
```

“Hard-coded” input and output files

Making more general-purpose rules: wildcards

- Wildcards ≈ Snakemake "variables"

```
rule example:
    input:
        'data/first_step.tsv'
    output:
        'results/first_step.txt'
    shell:
        'cp {input} {output}'
```

“Hard-coded” input and output files

```
rule example:
    input:
        'data/{sample}.tsv'
    output:
        'results/{sample}.txt'
    shell:
        'cp {input} {output}'
```

“General” input/output files
with wildcards

Making more general-purpose rules: wildcards

- Wildcards ≈ Snakemake "variables"

```
rule example:
    input:
        'data/first_step.tsv'
    output:
        'results/first_step.txt'
    shell:
        'cp {input} {output}'
```

“Hard-coded” input and output files

```
rule example:
    input:
        'data/{sample}.tsv'
    output:
        'results/{sample}.txt'
    shell:
        'cp {input} {output}'
```

“General” input/output files
with wildcards

- Enclose wildcard name with curly brackets { }

- How does Snakemake execution work when there are wildcards?

Making more general-purpose rules: wildcards

- Wildcards ≈ Snakemake "variables"

- Wildcards are “resolved” from the target and propagated to other directives
  - Regular expression matching: .+
    - ‘1 or more occurrences of any character except newline’
    - Can be constrained
  - **Using wildcards forces to ask for output(s)**: Snakemake doesn’t guess!
    - Target rules may not contain wildcards.

```
rule example:
    input:
        'data/{sample}.tsv'
    output:
        'results/{sample}.txt'
    shell:
        'cp {input} {output}'
```

```
snakemake --cores 1 results/first_step.txt
```

Snakemake interpretation:

`{sample} = "first_step"`

Making more general-purpose rules: wildcards

- Wildcards ≈ Snakemake "variables"

- Wildcards are “resolved” from the target and propagated to other directives
  - Regular expression matching: `.+`

- Both a workflow and a rule can use multiple wildcards

```
snakemake --cores 1 results/first_step.txt
```

Snakemake interpretation:

```
{sample} = "first"
{treatment} = "step"
```

```
rule example:
    input:
        'data/{sample}_{treatment}.tsv'
    output:
        'results/{sample}_{treatment}.txt'
    shell:
        'echo {wildcards.sample};'
        'cp {input} {output}'
```

Making more general-purpose rules: wildcards

- Wildcards ≈ Snakemake "variables"
- Wildcards are “resolved” from the target and propagated to other directives
  - Regular expression matching: `.+`
- Both a workflow and a rule can use multiple wildcards
- Input and output files do not have to share the same wildcards
- **All outputs/logs… created by a rule must have same wildcards!**

```
rule example:
    input:
        'data/{sample}.tsv'
    output:
        'results/{sample}_{treatment}.txt'
    shell:
        'echo {wildcards.sample};'
        'cp {input} {output}'
```

```
snakemake --cores 1 results/first_step.txt
```

Snakemake interpretation:

input = ‘data/first.tsv’

Creating a rule with multiple inputs/outputs

- Rules can use multiple inputs/outputs

Creating a rule with multiple inputs/outputs

- Rules can use multiple inputs/outputs

- Don’t forget the comma!

```
rule example:
    input:
        'data/first_step1.tsv',
        'data/first_step2.tsv'
    output:
        'results/first_step.txt'
    shell:
        'cat {input} > {output}'
```

Creating a rule with multiple inputs/outputs

- Rules can use multiple inputs/outputs

- Don’t forget the comma!

Input directive values are **unpacked** (replaced by a space-separated list)

```
rule example:
    input:
        'data/first_step1.tsv',
        'data/first_step2.tsv'
    output:
        'results/first_step.txt'
    shell:
        'cat {input} > {output}'
```

```
shell:
    cat data/first_step_1.tsv data/first_step_2.tsv > results/first_step.txt
```

Creating a rule with multiple inputs/outputs

- Rules can use multiple inputs/outputs

- Don’t forget the comma/semicolon!

- Inputs can be accessed by their positional index: *input[n]*
    - **Numbering starts at 0**

```
rule example:
    input:
        'data/first_step1.tsv',
        'data/first_step2.tsv'
    output:
        'results/first_step.txt'
    shell:
        'cat {input[0]} > {output};'
        'cat {input[1]} >> {output}'
```

Commands are concatenated

Creating a rule with multiple inputs/outputs

- Rules can use multiple inputs/outputs

- Don't forget the comma!

- Inputs can be accessed by their positional index: *input[n]*
    - **Numbering starts at 0**

```
rule example:
    input:
        'data/first_step1.tsv',
        'data/first_step2.tsv'
    output:
        'results/first_step.txt'
    shell:
        '''
        cat {input[0]} > {output}
        cat {input[1]} >> {output}
        '''
```

Creating a rule with multiple inputs/outputs

- Rules can use multiple inputs/outputs

- Don’t forget the comma!

- Inputs can be accessed by their positional index: *input[n]*
    - **Numbering starts at 0**

- Named input can be accessed by their names: *input.input_name*
    - You cannot mix named and unnamed inputs

```
rule example:
    input:
        input_1='data/first_step1.tsv',
        input_2='data/first_step2.tsv'
    output:
        'results/first_step.txt'
    shell:
        '''

        cat {input.input_1} > {output}
        cat {input.input_2} >> {output}
        '''
```

Creating a rule with multiple inputs/outputs

- Outputs work just like inputs
    - Separated by ‘,’
    - Can be named
    - Can be accessed by positional index or by name

- All outputs need to be generated or the job will fail

```
rule example:
    input:
        input_1='data/first_step1.tsv',
        input_2='data/first_step2.tsv'
    output:
        output_1='results/first_step1.txt',
        output_2='results/first_step2.txt'
    shell:
        '''

        cat {input.input_1} > {output.output_1}
        cat {input.input_2} > {output.output_2}
        '''
```

```
snakemake --cores 1 results/first_step_1.txt
```

→ results/first_step_1.txt, results/first_step_2.txt

Processing list of files: the expand syntax

- expand(): Snakemake function to automatically expand a wildcard expression to several wildcard values
  - Useful to define multiple inputs or outputs with a common pattern

Processing list of files: the expand syntax

- expand(): Snakemake function to automatically expand a wildcard expression to several wildcard values
    - Useful to define multiple inputs or outputs with a common pattern
    - Syntax: expand('{wildcard_name}', wildcard_name=<values>)
        - <values>: iterable (*i.e.* list, tuple, set) containing the wildcard values

```
rule example:
    input:
        'data/A.tsv',
        'data/B.tsv',
        'data/C.tsv'
    output:
        'results/total.tsv'
    shell:
        'cat {input} > {output}'
```

```
rule example:
    input:
        expand('data/{sample}.tsv', sample=[‘A,‘B’,‘C’])
    output:
        'results/total.tsv'
    shell:
        'cat {input} > {output}'
```

➢ The rule example uses all three input files to generate a single output file. expand() does not apply the rule separately to the three inputs!

Processing list of files: the expand syntax

- When there are several wildcards, expand() creates all possible combinations

Processing list of files: the expand syntax

- When there are several wildcards, expand() creates all possible combinations

```
samples=[‘A,‘B’]
replicates = [1, 2]

rule example:
    input:
        expand('data/{sample}_{replicate}.tsv', sample=samples, replicate=replicates)
    output:
        'results/total.tsv'
    shell:
        'cat {input} > {output}'
```

➢ input = 'data/A_1.tsv data/A_2.tsv data/B_1.tsv data/B_2.tsv'

Processing list of files: the expand syntax

- The wildcards in expand are INDEPENDENT from any other wildcard in the rule

Processing list of files: the expand syntax

- The wildcards in expand are INDEPENDENT from any other wildcard in the rule

```
samples=[‘A,‘B’]
replicates = [1, 2]

rule example:
    input:
        expand('data/{sample}_{replicate}.tsv', sample=samples, replicate=replicates)
    output:
        'results/{sample}.tsv'
    shell:
        'cat {input} > {output}'
```

➢ In this case, the value of the {sample} wildcard will NOT be propagated to the input

Optimising workflow performances

- Producing log files
- Benchmarking rules
- Multi-threading and controlling resource usage

Optimising workflow performances: log files

- ‘log’ is a directive; its value is a path to a log file for a rule

Optimising workflow performances: log files

- ‘log’ is a directive; its value is a path to a log file for a rule
    - Can be accessed with a placeholder in ‘shell’: {log}

- Logs still need to be handled manually for each command, but Snakemake automatically creates the directory in the log file path

```
rule example:
    input:
        'data/first_step.tsv'
    output:
        'results/first_step.txt'
    log:
        'logs/first_step.log'
    shell:
        'cp {input} {output} 2> {log}'
```

Optimising workflow performances: log files

- ‘log’ is a directive; its value is a path to a log file for a rule
  - Can be accessed with a placeholder in ‘shell’: {log}
- Logs still need to be handled manually for each command, but Snakemake automatically creates the directory in the log file path
- Log files must have the **same wildcards as the output**!
- Best to regroup logs in a ‘logs’ folder

```
rule example:
    input:
        'data/first_step.tsv'
    output:
        'results/first_step.txt'
    log:
        'logs/first_step.log'
    shell:
        'cp {input} {output} 2> {log}'
```

Optimising workflow performances: benchmarks

- ‘benchmark’ is a directive; its value is a path to a benchmark results file for a rule

Optimising workflow performances: benchmarks

- ‘benchmark’ is a directive; its value is a path to a benchmark results file for a rule

- Snakemake will automatically measure runtime and memory usage for the rule and save it to the file

```
rule example:
    input:
        'data/first_step.tsv'
    output:
        'results/first_step.txt'
    benchmark:
        'benchmarks/first_step.txt'
    shell:
        'cp {input} {output}'
```

Optimising workflow performances: benchmarks

- ‘benchmark’ is a directive; its value is a path to a benchmark results file for a rule
- Snakemake will automatically measure runtime and memory usage for the rule and save it to the file
- Benchmark files must have the **same wildcards as the output**!
- Best to regroup benchmarks in a ‘benchmarks folder

```
rule example:
    input:
        'data/first_step.tsv'
    output:
        'results/first_step.txt'
    benchmark:
        'benchmarks/first_step.txt'
    shell:
        'cp {input} {output}'
```

Optimising workflow performances: threads

- ‘threads’ is a directive; its value is the number of threads to allocate to each job spawned by a rule
    - New kind of directive value: numeric (integer)
    - **Check whether software can actually multithread!**

Optimising workflow performances: threads

- ‘threads’ is a directive; its value is the number of threads to allocate to each job spawned by a rule
  - New kind of directive value: numeric (integer)
  - **Check whether software can actually multithread!**

```
rule example:
    input:
        'data/first_step.tsv'
    output:
        'results/first_step.txt'
    threads: 4
    shell:
        'command --threads {threads} {input} > {output}'
```

  - In local mode, total number of threads allocated to Snakemake is constrained by the execution parameter ‘--cores’

Optimising workflow performances: resources

- ‘resources’ is a directive; its values aim to set the resources available for a job
    - New kind of directive value: pair of <key>=<value>

Optimising workflow performances: resources

- ‘resources’ is a directive; its values aim to set the resources available for a job
  - New kind of directive value: pair of <key>=<value>
- mem_<unit>
  - Amount of memory needed by the job
  - <unit>: mb, gb, tb…
- runtime_<unit>
  - Amount of wall clock time a job needs to run
  - <unit>: s, m, h, d…

```
rule example:
    input:
        'data/first_step.tsv'
    output:
        'results/first_step.txt'
    resources:
        mem_gb=1,
        runtime_s=3600
    shell:
        'command {input} > {output}'
```

Exercises

- Through the day:
  - Develop a simple RNAseq analysis workflow, from reads (fastq files) to Differentially Expressed Genes (DEG)

- For now:
  - Session 2:
    - Use multiple inputs and outputs
    - Use placeholders and wildcards
    - Optimise workflow performance
    - Visualise a DAG
  - Session 3:
    - Use non-file parameters
    - Manage non-conventional outputs
    - Process list of inputs
    - Modularise a workflow