Snakemake Workshop

Aleksei Dievskii, JetBrains Research

Installation

```
$ cd snakemake-workshop/conda

$ conda env create --file snakemake.yaml --name smk

$ conda activate smk
```

(or `$ source activate smk`)

```
$ snakemake -v
```

What is Snakemake

Snakemake is a workflow management system.

- intended to be reproducible and platform-independent (runs on a workstation, a server, a cluster, in cloud etc);
- able to use `conda` environments;
- smart concurrency and dependency management;
- inspired by GNU `make`.

Workshop outline

Construct a workflow to:

- **sort** several BAM files,
- **merge** them, and
- **remove duplicates.**

Task 01

Task01: sort a BAM file

```
$ cd task01
```

We want to coordinate-sort `A_unsorted.bam` -> `A_sorted.bam`.

Bash way: `$ samtools sort -o A_sorted.bam A_unsorted.bam`

Snakemake way:

```
rule sort_bam_A:
    input: "A_unsorted.bam"
    output: "A_sorted.bam"
    shell: 'samtools sort -o A_sorted.bam A_unsorted.bam'
```

Task01: basic structure

Workflow description is in a file named `Snakefile`.

`Snakefile` is composed mostly of rule declarations.

Rules describe how to make the output from the input.

Run

```
$ snakemake
```

in the folder `task01`.

Task 02

Task02: sort and merge two BAM files

```
$ cd task02
```

Sort and merge the two BAM files.

Look at the `Snakefile`. Note: `merge_bams` rule has two inputs.

Run

```
$ snakemake
```

What do you think should happen?

Task02: sort and merge two BAM files

Only `sort_bam_A` got processed.

Snakemake doesn't know which rules you need!

It **targets the first one** in the file by default.

You can override this by calling

```
$ snakemake <target rule name>
```

Task02: DAG of jobs

- start with **the target rule**
- match current rule's **input** to another rule's **output**
- if **none match,** input must be present as **initial data**
- **continue** until the graph is built
- **launch** jobs in an appropriate order

In our case, `merge_bams` depends on `sort_bam_A` and `sort_bam_B`.

If we target `merge_bams`, Snakemake will first complete the `sort` jobs.

Task02: do what needs to be done

Either move `merge_bams` **to the top** or **target it** specifically.

How many jobs were launched? Why?

Task02: do what needs to be done

Either move `merge_bams` **to the top** or **target it** specifically.

How many jobs were launched? Why?

`A_sorted.bam` **already exists and is newer** than `A_unsorted.bam`.

Let's investigate:

```
$ touch A_unsorted.bam
```

```
$ snakemake
```

Task 03

Task03: sort and merge even more BAMs

```
$ cd task03
```

Sort and merge the three BAM files.

Look at the `Snakefile`. Do you like it? Launch it anyway.

Task03: sort and merge even more BAMs

```
$ cd task03
```

Sort and merge the three BAM files.

Look at the `Snakefile`. Do you like it? Launch it anyway.

Copy-paste typo! `sort_bam_C` hasn't created output.

Workflow is **terminated** and output is **removed** if the job has **non-zero exit status** or **fails to produce its declared output.**

Task03: fail fast and unofficial Bash strict mode

"Fail fast" paradigm: **the workflow is terminated** as soon as **any job fails.**

Already running jobs are allowed to finish.

Each shell command is executed in **unofficial Bash strict mode**:

```
$ set -euo pipefail;
```

- `-e`: fail if **any command in the script** fails;
- `-u`: fail on **undefined variable** reference;
- `-o pipefail`: a pipe fails if **any command in the pipe** fails.

Task03: substitutions

Let's reduce the copy-paste errors.

Snakemake strings are implicitly formatted (as in Python's `str.format`).

Use variable substitutions!

Replace the `shell` string in `sort_bam` rules with:

```
shell: 'samtools sort -o {output} {input}'
```

Names in curly braces are replaced with **values**.

Task03: corrupted data

Caution: `sort_bam_C` wrote to `B_sorted.bam`.

How to tell Snakemake it needs to be recalculated?

Task03: corrupted data

Caution: `sort_bam_C` wrote to `B_sorted.bam`.

How to tell Snakemake it needs to be recalculated? Either:

- `$ rm B_sorted.bam`, or
- `$ touch B_unsorted.bam`, or
- `$ snakemake --forcerun sort_bam_B`, or
- `$ snakemake --forceall`.

Task03: wildcards

The amount of copy-paste is still too high!

Replace `sort_bam_A` rule with:

```
rule sort_bam:
    input: "{sample}_unsorted.bam"
    output: "{sample}_sorted.bam"
    shell: 'samtools sort -o {output} {input}'
```

Discard the remaining `sort` rules. Run Snakemake with `--forceall`.

Task03: rules and jobs

Snakemake tries to match the wildcards.

The `merge_bams` inputs can be matched to `sort_bam` output. **One** `sort_bam` **rule** generates **three** `sort_bam` **jobs**.

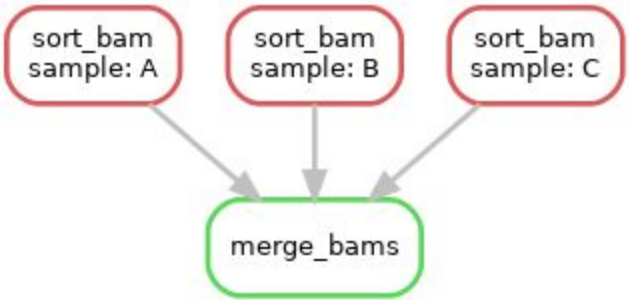
With `graphviz`, look at the job graph using:

```
$ snakemake --dag | dot | display
```

or:

```
$ snakemake --dag | dot -Tpdf > job_dag.pdf
```

Task03: DAG of jobs



Task03: wildcard notes

Wildcards aren't allowed in **the target rule.**

Multiple matches -> Snakemake fails.

This can be avoided by:

- rule precedence (`ruleorder: rule1 > rule2`)
- match ambiguity (`snakemake --allow-ambiguity`)
- wildcard pattern (`"GSM{id, [0-9]+}.bam"`)

Task 04

Task04: concurrency

Sort and merge 10 BAM files.

Modern CPUs are **multi-core.** Why waste time and limit ourselves to **one thread**?

Run `$ snakemake --cores 4`.

Task04: concurrency

Sort and merge 10 BAM files.

Modern CPUs are **multi-core.** Why waste time and limit ourselves to **one thread**?

Run `$ snakemake --cores 4`.

Independent jobs can be run concurrently.

Snakemake can also run on a cluster or in the cloud.

Task04: threads

By default, each job claims one thread. This can be overridden:

```
rule sort_bam:
    input: "{sample}_unsorted.bam"
    output: "{sample}_sorted.bam"
    threads: 2
    shell: 'samtools sort -@ {threads} -o {output} {input}'
```

Each `sort_bam` job will use **up to two** threads (maybe fewer!).

```
$ snakemake --cores 4 --forceall
```

Task04: temporary files

`_sorted` files are cluttering the folder!

Replace `sort_bam` output with:

```
    output: temp("{sample}_sorted.bam")
```

```
$ snakemake --forceall
```

`temp` outputs are **removed** as soon as **the last job** requiring them finishes.

Task 05

Task05: remove duplicates

Sort the BAMs, merge them and remove duplicates.

We're going to use `picard MarkDuplicates`.

`picard` is a complex tool with unintuitive syntax. We'll use a wrapper.

Look at the `Snakefile`. Note: `remove_duplicates` rule has `wrapper` instead of `shell`.

```
$ snakemake -n
```

This performs a **dry run** (print the jobs, don't launch them).

Task05: wrappers

Special curated BitBucket wrapper repository (https://bitbucket.org/snakemake/snakemake-wrappers/)

You can write and supply your own wrappers.

Add `--use-conda` to command-line options.

```
$ snakemake --use-conda
```

Picard was automatically installed in a Conda environment.

Task05: params and logs

`params` is an auxiliary rule property for non-file execution parameters.

`log` property declares a log file. Log files can be matched to inputs.

Why not just declare it as output?

Task05: params and logs

`params` is an auxiliary rule property for non-file execution parameters.

`log` property declares a log file. Log files can be matched to inputs.

Why not just declare it as output?

Snakemake **deletes the output** if the rule fails. But **the log persists!**

Task05: more wrappers

Wrappers increase reproducibility! Use them whenever possible.

In `sort_bam` rule, `shell` can be replaced with:

```
    wrapper: "0.31.1/bio/samtools/sort"
```

In `merge_bams` rule, `shell` can be replaced with:

```
    wrapper: "0.31.1/bio/samtools/merge"
```

Miscellaneous

Python and R interop

Snakemake is a Python dialect. Python code can be written in a `Snakefile`.

Instead of `shell` or `wrapper`, you can use `script` or `run`.

```
    run:
        <arbitrary Python code>
```

or

```
    script: "path/to/script.py"
```

Snakemake can run `R` scripts. Global and local properties are available from inside the script via a `snakemake` object.

Assorted rule properties

- `shadow`: launch the job in an isolated folder
- `conda`: specify a YAML file with conda environment for the job
- `resources`: specify the required resources for the job (e.g. RAM)
- `message`: print a message to stdout
- `priority`: prioritize a job
- `protected`: the output is write-protected after the job finishes
- `dynamic`: the number of output files is not known until the job finishes
- `directory`: the output is a directory
- `pipe`: the output is a named pipe

Assorted command-line options

- `-r`: print the reason for each rule's execution
- `-p`: print the shell commands of each job
- `-s`: use a custom-named Snakefile
- `--cluster`: launch on a computational cluster
- `--kubernetes`: launch in Kubernetes
- `--config`: pass a list of key-value pairs accessible via config object
- `--configfile`: read the config from a JSON or YAML file
- `-d`: set working directory
- `-k`: don't terminate the workflow if one job fails

Thanks for the attention!