Snakemake 001

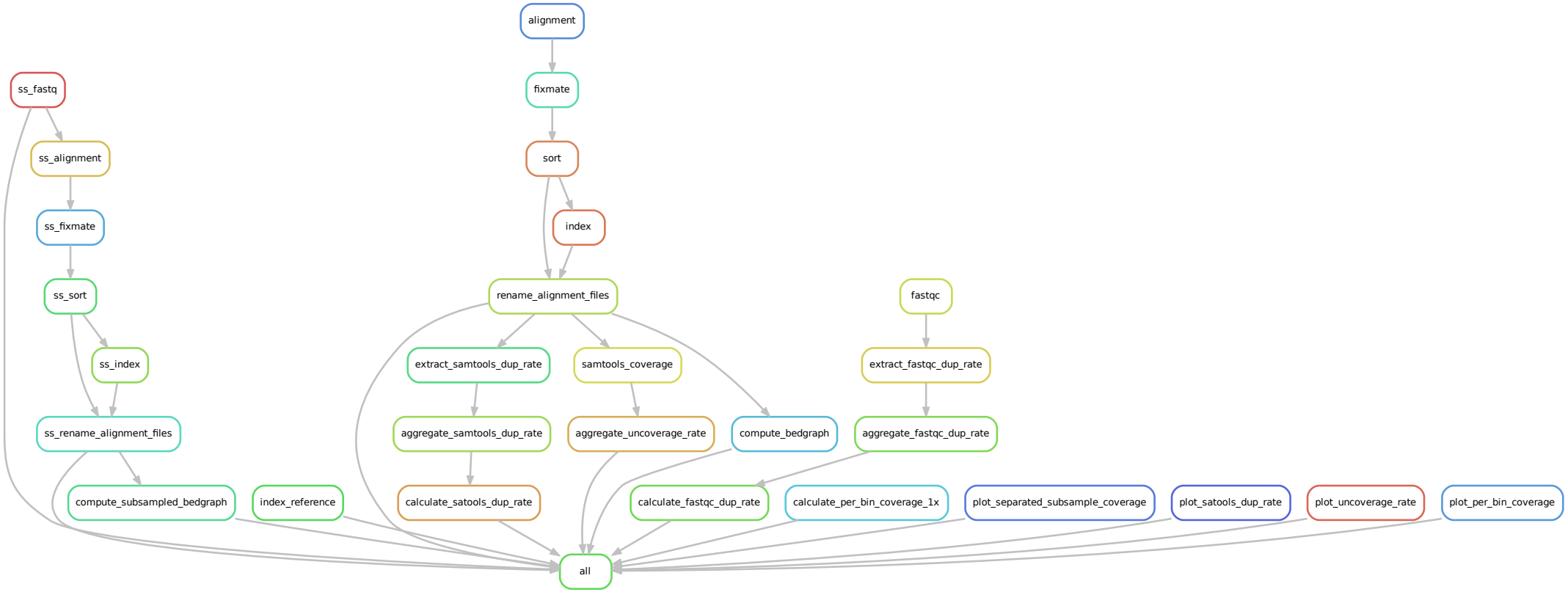
Zhipeng Zhang

03/05/2023

Introduction

- *The Snakemake workflow management system is a tool to create **reproducible and scalable** data analyses. Workflows are described via a human readable, **Python** based language. They can be seamlessly scaled to server, **cluster**, grid and cloud environments, without the need to modify the workflow definition*[1].

A Snakemake Pipeline



Installation

- It is recommended that a `conda` environment to be set up as it is very useful as a package management tool (we'll see later). See this link from the GMS training page (https://whg-training.github.io/whg-training-resources/prerequisites/CONDA/) of how this can be achieved. Then, one can run

```
mamba install snakemake
```

- Alternatively, one could load snakemake on the cluster[2] using

```
module load snakemake/5.26.1-foss-2019b-Python-3.7.4
```

Structure

- A (suggested) tree layout of a snakemake folder can look like

```
snakemake/
  data/
    # reads and reference
  pipelines/
    # snakemake files and config files
  results/
    # results
```

Though it is also at your discretion[3][4].

Config File

- A config file can be seen as a summary of the original data we have, which can look like

```
{
  “ref”: “data/reference/HRCh38.fa”,
  “samples_1x”: [
    {“name”: “alpha”, “ID”: “sample1”},
    {“name”: “alpha”, “ID”: “sample2”},
    …
  ]
  “samples_20x”: “samples_20x.tsv”
}
```

And then the .tsv file can be read in snakemake using `pandas`.

Load Data

- Suppose our main snakemake file is named `master.smk`, we can load the data by first specify the config file and then read it using the Pythonic way:

```
configfile: “pipelines/config.json”
import pandas as pd

samples_1x = [sample[“name”] for sample in config[samples_1x]]
samples_20x = pd.read_table(“samples_20x.tsv”)
```

And then the .tsv file can be read in snakemake using `pandas`.

Caveats 1

- The reason that a conda environment is recommended is because we need to manually load lots of packages or modules (e.g.: `pandas`) , which would be rather simple if we have a package management stuff.
  - Solution: Use conda!
- Whether or not using commas and/or indentation errors can be extremely annoying, as the pipeline won't run and the error message from snakemake is usually uninformative.
  - Solution: Commas should be used in the config file and snakemake rules but not elsewhere.
  - Solution: Since VSCode may be tricky in terms of indentations, using terminal editors (`nano, vim`, etc.) to debug can be very helpful.

Snakemake Rule 1: Basics

- A rule defines a snakemake job that it is supposed to run. It should include an input, output, and shell command:

```
rule index_reference:
  input:
    ref = config[“ref”]
  output:
    bwt = “data/reference/GRCh38.fa.bwt”
  shell: """
    bwa index {input.reference}
  """
```

This rule specifies the input file (input), output to be expected (output), and what command to run (shell).

Snakemake Rule 1: Basics

- All desired outputs from snakemake files should be specified in a specific rule called `rule all`:

```
rule all:
  input:
    bwt = “data/reference/GRCh38.fa.bwt”,
    …
```

which helps snakemake to decide what rules and in which order they need to be run.

Caveats 2

- The following layout is suggested:

<master.smk>
```
configfile: “pipelines/config.json”
include: “index.smk”

rule all:
  input:
    bwt = “data/reference/GRCh38.fa.bwt”
```

<index.smk>
```
configfile: “pipelines/config.json”
rule index_reference:
  input:
    ref = config[“ref”]
  output:
    bwt = “data/reference/
    GRCh38.fa.bwt”
  shell: """
    bwa index {input.reference}
  """
```

Execution (Local)

- Following the previous example, you can type the following command in the terminal to run snakemake:

```
snakemake -s pipelines/master.smk -c 1
```

It is recommended that we run snakemake from the top-level folder. Option -c (number of cpus) has to be specified for snakemake to run.

Before any real execution of the pipeline, you can first dry-run it by

```
snakemake -s pipelines/master.smk -n
```

to check if errors present (though not always useful).

Drawing DAGs

- One can use the following command to draw a DAG (see slide 3):

```
snakemake -s pipelines/master.smk --forceall --dag | \
dot -Tpdf > dag.pdf
```

Or the following for a succinct DAG:

```
snakemake -s pipelines/master.smk --forceall –rulegraph | \
dot -Tpdf > dag.pdf
```

Snakemake Rule 2: Wildcards

- Wildcards render snakemake scalability. One can ask a snakemake rule to run on a bunch of samples using wildcards by:

```
rule alignment:
  input:
    fastq1 = “data/samples/{name}_1.fastq”,
    fastq2 = “data/samples/{name}_2.fastq”,
    ref = config[“ref”]
  output:
    bam = “results/aligned/{name}.bam”
  shell: """
    bwa mem -o {input.fastq} {input.ref} {input.fastq1}
    {input.fastq2}
  """
```

Snakemake Rule 2: Wildcards

Since we have two samples as specified in our config file, we need the following `rule all`:

```
<master.smk>
samples_1x = [sample[“name”] for sample in config[samples_1x]]

rule all:
  input:
    bams = expand(“results/aligned/{name}.bam”, name = samples_1x)
```

Snakemake Rule 3: Additional Features

- `Params, resources, and threads` are stuff that I found sometimes useful and/or necessary to specify in a snakemake rule:

```
rule fastqc:
  input:
    fastq1 = “data/samples/{name}_1.fastq”,
    fastq2 = “data/samples/{name}_2.fastq”
  output:
    html1 = “results/qc/{name}_1_fastqc.html”,
    html2 = “results/qc/{name}_2_fastqc.html”,
    zip1 = temp(“results/qc/{name}_1_fastqc.zip”),
    zip2 = temp(“results/qc/{name}_2_fastqc.zip”)
  params: outputdir = “results/qc/”
  # mem_mb: 1000
  # threads: 1
  shell: """
    fastqc -q -o {params.outputdir} {input.fastq1} {input.fastq2}
  """
```

Snakemake Rule 4: Advanced Features

- `Ruleorder` is one thing that I find very useful for snakemake to determine which rule to run if there are multiple rules that can generate same outputs:

```
num_samples, = glob_wildcards("data/samples/{name}_1.fastq")
if len(num_samples) >= 5:
    ruleorder: plot_aggregate > plot_separate
else:
    ruleorder: plot_separate > plot_aggregate

rule plot_separate:
  …
rule plot_aggregate:
  …
```

Snakemake Rule 4: Advanced Features

- `Ruleorder` is one thing that I find very useful for snakemake to determine which rule to run if there are multiple rules that can generate same outputs:

```
num_samples, = glob_wildcards("data/samples/{name}_1.fastq")
if len(num_samples) >= 5:
    ruleorder: plot_aggregate > plot_separate
else:
    ruleorder: plot_separate > plot_aggregate

rule plot_separate:
  …
rule plot_aggregate:
  …
```

Execution (Cluster)

- Snakemake execution on cluster is more tricky. Basically you need to have a bunch of files set up[5] (see https://github.com/jdblischak/smk-simple-slurm). Then, you could invoke snakemake by

```
snakemake -s pipelines/master.smk --profile slurm/
```

where you should put a `config.yaml` file under a subdirectory called `slurm`.

Alternatively, you can run it directly by

```
snakemake -s pipelines/master.smk -j 1 \
--default-resource 'mem_mb=100' \
--latency-wait 5 \
--max-jobs-per-second 0.01 \
--cluster 'sbatch -p short -c1 --mem {resources.mem_mb}'
```

Bibliography

[1] https://snakemake.readthedocs.io/en/stable/.
[2] https://www.medsci.ox.ac.uk/for-staff/resources/bmrc/scientific-software-directory#b.
[3] https://whg-training.github.io/whg-training-resources/next_generation_sequencing/building_an_ngs_pipeline/tips_and_tricks/.
[4] https://snakemake.readthedocs.io/en/stable/snakefiles/deployment.html.
[5] https://github.com/jdblischak/smk-simple-slurm.

Thank you :)