E4/E5 Bachelor Programmierkurs

Snakemake 

“A scalable bioinformatics workflow engine”

Tobias Cremer, Hendrik Speiser
(originally by Vukan Jevtić, Louis Gerken)

April 10, 2024

Technische Universität Dortmund

Snakemake installation

on your own device:

```
$ conda config --add channels bioconda
$ conda install snakemake
```

on our E4 workstations:

- no installation necessary
- do **NOT** load CVMFS software packages before using snakemake!

Cite as:

- Köster, Johannes and Rahmann, Sven. “Snakemake - A scalable bioinformatics workflow engine”. Bioinformatics 2012.

Further reading (links)

- Documentation
- snakemake for htcondor

Anatomy of a Snakefile

Workflow: `raw_data.csv` → `data.csv` → `plot.pdf`

Anatomy of a Snakefile

Workflow: `raw_data.csv` → `data.csv` → `plot.pdf`

Snakemake

```
rule make_plot:
    input:  "data.csv"
    output: "plot.pdf"
    shell:  "python plot.py"

rule select_data:
    input:  "raw_data.csv"
    output: "data.csv"
    shell:  "python selection.py"
```

Minimal console command

```
$ snakemake
```

Anatomy of a Snakefile

Workflow: `raw_data.csv` → `data.csv` → `plot.pdf`

Snakemake

```
rule make_plot:
    input:  "data.csv"
    output: "plot.pdf"
    shell:  "python plot.py"

rule select_data:
    input:  "raw_data.csv"
    output: "data.csv"
    shell:  "python selection.py"
```

Running by specifying output file

```
$ snakemake plot.pdf
```

Anatomy of a Snakefile

Workflow: `raw_data.csv` → `data.csv` → `plot.pdf`

Snakemake

```
rule make_plot:
    input:  "data.csv"
    output: "plot.pdf"
    shell:  "python plot.py"

rule select_data:
    input:  "raw_data.csv"
    output: "data.csv"
    shell:  "python selection.py"
```

Running by specifying name of rule

```
$ snakemake make_plot
```

Anatomy of a Snakefile

Workflow: `raw_data.csv` → `data.csv` → `plot.pdf`

Snakemake

```
rule make_plot:
    input:  "data.csv"
    output: "plot.pdf"
    shell:  "python plot.py"

rule select_data:
    input:  "raw_data.csv"
    output: "data.csv"
    shell:  "python selection.py"
```

Output Input Recipe

Running snakemake on our workstations

To run the snakemake command you always have to define the number of CPU to use:

```
$ snakemake -j1
```

“snakemake -nr” console output (dry run with reason for each rule)

```
Building DAG of jobs...
Job counts:
      count   jobs
      1       make_plot
      1       select_data
      2

[Wed Feb 19 15:50:05 2020]
rule select_data:
  input: raw_data.csv
  output: data.csv
  jobid: 1
  reason: Missing output files: data.csv


[Wed Feb 19 15:50:05 2020]
rule make_plot:
  input: data.csv
  output: plot.pdf
  jobid: 0
  reason: Missing output files: plot.pdf; Input files updated by another job: data.csv

Job counts:
      count   jobs
      1       make_plot
      1       select_data
      2
This was a dry-run (flag -n). The order of jobs does not reflect the order of execution.
```

“snakemake” console output

```
Building DAG of jobs...
Using shell: /usr/local/bin/bash
Provided cores: 256
Rules claiming more threads will be scaled down.
Job counts:
        count   jobs
        1       make_plot
        1       select_data
        2

[Wed Feb 19 15:38:12 2020]
rule select_data:
    input: raw_data.csv
    output: data.csv
    jobid: 1

[Wed Feb 19 15:38:12 2020]
Finished job 1.
1 of 2 steps (50%) done

[Wed Feb 19 15:38:12 2020]
rule make_plot:
    input: data.csv
    output: plot.pdf
    jobid: 0

[Wed Feb 19 15:38:14 2020]
Finished job 0.
2 of 2 steps (100%) done
Complete log: /net/nfshome/home/somepath/.snakemake/log/2020-02-19T153812.433826.snakemake.log
```

Physics analysis example

To analyse data, we usually have a raw data file and some data processing steps

```
rule plot_results:
    input:
      "data_selected.root",
      "plot.py"
    output: "results.pdf"
    shell: "python plot.py data_selected.root"

rule data_selection:
    input:  "data_preprocessed.root"
    output: "data_selected.root"
    shell: "python selection.py data_preprocessed.root data_selected.root"

rule data_preprocessing:
    input:  "data_raw.root"
    output: "data_preprocessed.root"
    shell:  "python analysis.py data_raw.root data_preprocessed.root"
```

`input`, `output`, `shell` etc. are optional

Physics analysis example

To analyse data, we usually have a raw data file and some data processing steps

```
rule plot_results:
    input:
      "data_selected.root",
      "plot.py"
    output: "results.pdf"
    shell: "python plot.py data_selected.root"

rule data_selection:
    input:  "data_preprocessed.root"
    output: "data_selected.root"
    shell: "python selection.py data_preprocessed.root data_selected.root"

rule data_preprocessing:
    input:  "data_raw.root"
    output: "data_preprocessed.root"
    shell:  "python analysis.py data_raw.root data_preprocessed.root"
```

Would be nice to reduce amount of repetitions

Physics analysis example

We can alias files ⇒ rules can reference their own parameters

```
rule plot_results:
    input:
      data = rules.data_selection.output,
      py_file = "plot.py"
    output: "results.pdf"
    shell: "python {input.py_file} {input.data}"

rule data_selection:
    input:  "data_preprocessed.root"
    output: "data_selected.root"
    shell: "python selection.py {input} {output}"

rule data_preprocessing:
    input:  "data_raw.root"
    output: "data_preprocessed.root"
    shell:  "python analysis.py {input} {output}"
```

Strings containing `{...}` are formatted

Executing arbitrary python code in Snakefiles

A Snakefile can be treated almost like a python script:

```
import uproot
import pandas
import numpy as np

def say_hello(name):
    print(f"Hello {name}!")

rule somerule:
    input: files = [f"dataset_{num}.root" for num in range(100)]
    run:
        say_hello("E4")
        for tfile in input.files:
            ds = uproot.open(tfile)["DecayTree"]
            data = ds.arrays("B_P[XY]", outputtype=pandas.DataFrame)
            print(np.sqrt(data.B_PX**2 + data.B_PY**2))
```

Executing python scripts

Instead of `shell` or `run` a script can be invoked.
(It does not need to be a python script)

```
rule massfit:
    input: "data.root"
    output: "parameters.txt", "plot.pdf"
    params:
      fitConstrained = False,
      extendedMLFit = True
    script: "massfit.py"
```

massfit.py:

```
import ROOT as R
from ROOT import RooFit
fitContrained = snakemake.params.fitConstrained
extendedMLFit = snakemake.params.extendedMLFit
# Load datasets, fit something...
```

Useful command line options

Just print scheduled rules without running

```
$ snakemake <rule> -n
```

Print the reason for running each rule as well

```
$ snakemake <rule> -n -r
```

Force execution of target

```
$ snakemake <rule> -f
```

Force execution of a target and its workflow

```
$ snakemake <rule> -F
```

Force re-execution of rule and its workflow

```
$ snakemake <rule> -R
```

Run workflow until specified rule

```
$ snakemake <rule> --until <rule>
```

Update timestamps →force files up to date

```
$ snakemake <rule> --touch
```

Ignore errors

```
$ snakemake <rule> --keep-going
```

Rerun incomplete rules (in case of crash)

```
$ snakemake --rerun-incomplete
```

Print shell commands that snakemake runs

```
$ snakemake -p
```

Abstraction of input and output

```
1    rule file_requester:
2        input: "file_A_0.txt",
3        input: "file_A_1.txt",
4        input: "file_A_2.txt",
5        input: "file_B_0.txt",
6        input: "file_B_1.txt",
7        input: "file_B_2.txt",
```

- EXPAND(...)
- WILDCARDS

expand(...)

Snakemake has an integrated method for generating lists of files: `expand(...)`

```
1 rule file_requester:
2     input: expand("file_{cat}_{num}.txt", cat=["A", "B"], num=range(3))
```

The following list is created as input:

`file_A_0.txt, file_A_1.txt, file_A_2.txt, file_B_0.txt, file_B_1.txt, file_B_2.txt`

Next level of abstraction: Wildcards

A wildcard rule matches patterns in dependencies

```
rule single_selection:
    input:  "data_{num}.root"
    output: "data_{num}_selected.root"
    shell:  "python run_selection.py {input} {output}"

rule select_files:
    input: expand("data_{n}_selected.root", n=range(10))
```

Note:

1. Input and output *must* contain same wildcards
2. A wildcard rule cannot be called directly by its name
3. Two rules should not contain the same outputs

Next level of abstraction: Wildcards

A wildcard rule matches patterns in dependencies

```
rule single_selection:
    input:  "data_{num}.root"
    output: "data_{num}_selected.root"
    shell:  "python run_selection.py {input} {output}"

rule select_files:
    input: expand("data_{n}_selected.root", n=range(10))
```

If one runs

```
$ snakemake select_files
```

rule `select_files` is going to call the wildcard rule for 10 different files

Next level of abstraction: Wildcards

A wildcard rule matches patterns in dependencies

```
rule single_selection:
    input:  "data_{num}.root"
    output: "data_{num}_selected.root"
    shell:  "python run_selection.py {input} {output}"

rule select_files:
    input: expand("data_{n}_selected.root", n=range(10))
```

If one runs

```
$ snakemake data_7_selected.root
```

rule `select_files` is going to call the wildcard rule for case `num = 7`

Wildcard constraints

But what if a certain combination of wildcards needs to be treated differently?

⇒ Use wildcard constraints

```
rule somerule:
    input:  "data_{year}.root"
    output: "massplot_{year}.pdf"
    wildcard_constraints: year="201[578]"
    shell: "python massfit.py {input}"

rule somerule_special_case:
    input:  rules.somerule.input
    output: rules.somerule.output
    wildcard_constraints: year="2016"
    shell: "python massfit.py {input} -be_careful"
```

If you need to treat a wildcard value differently from the others, you need to constrain them for each relevant rule as shown here

Here, regex can be quite useful: regex101.com.

Parallelizing everything!

At some point, you may realize that you need more than 1 CPU...
Luckily, there is an option for that:

```
$ snakemake my_analysis -j20
```

This command is going to (try to) parallelize your workflow into 20 parallel chains

Parallelizing everything!

At some point, you may realize that you need more than 1 CPU...
Luckily, there is an option for that:

```
$ snakemake my_analysis -j20
```

This command is going to (try to) parallelize your workflow into 20 parallel chains

But what do you do if you need $\mathcal{O}(100)$ CPUs, for example 300 CPUs and 1TB of RAM?
⇒ Send your jobs to our own cluster! (For huge jobs, please ask your supervisor for permission)

Parallelizing everything!

At some point, you may realize that you need more than 1 CPU...
Luckily, there is an option for that:

```
$ snakemake my_analysis -j20
```

This command is going to (try to) parallelize your workflow into 20 parallel chains

But what do you do if you need $\mathcal{O}(100)$ CPUs, for example 300 CPUs and 1TB of RAM?
⇒ Send your jobs to our own cluster! (For huge jobs, please ask your supervisor for permission)
But what do you do if you need $\mathcal{O}(1000)$ CPUs and $\mathcal{O}(20)$ GPUs?
⇒ Send your jobs to the LiDO cluster of our university

Advanced Topics

Using snakemake to submit to a computing cluster

A tutorial can be found here: `Click me!`
Submitting to a HTCondor computing cluster can be as simple as:

```
$ snakemake <rule> -j500 --profile htcondor
```

Setting optional resource requirements for the scheduler

```
rule clusterrule:
    input: "file.txt"
    output: "outfile.txt"
    threads: 12
    resources:
      MaxRunHours=24,      # Job takes up to a day
      request_memory=1024  # Request RAM in MB
      request_gpus=1,      # Submit to a machine with GPU
      request_disk=1000000 # Disk requirement in kB
    run:
        print(f"This rule is allowed to use {threads} threads")
```

Submit with the same command lean back while the cluster takes off 🚀

Subworkflows

Snakefiles can be connected via subworkflows:

Main Snakefile

```
subworkflow another_worklow:
    workdir: 'path/to/other/workdir'
    snakefile: 'path/to/other/workdir/Snakefile'

rule master_rule:
    input: another_worklow("text.txt")
```

Another Snakefile

```
rule create_file:
    output: "text.txt"
    shell: "touch text.txt"
```

Some more useful tips

Various file wrappers:

- Timestamp of files wrapped in `ancient("filename")` is ignored
- Files wrapped in `protected("filename")` are not deleted by Snakemake
- A file wrapped in `temp("filename")` is deleted after rule is finished
- `touch("filename")` creates an empty file with that name as output

Setting a function as rule input:

```
1 def get_files(wildcards):
2     return #[ A list of files according to wildcards]
3
4 rule arule:
5     input: get_files
```

Some more useful tips

Various file wrappers:

- Timestamp of files wrapped in `ancient("filename")` is ignored
- Files wrapped in `protected("filename")` are not deleted by Snakemake
- A file wrapped in `temp("filename")` is deleted after rule is finished
- `touch("filename")` creates an empty file with that name as output

Using a config file (json or yaml) for yaml file →exercise

config.json

```
1 {
2   "param_a" : "362",
3   "param_b" : "cat"
4 }
```

Snakefile

```
1 configfile: "config.json"
2
3 param_a = config["param_a"]
4 param_b = config["param_b"]
```

Common Errors

Common mistakes: Wrong wildcard deduction

```
rule somerule:
    output: "afile_{year}_{polarity}.root"
    shell: "echo Running rule"

rule requester:
    input: "afile_2017_MagnetUp_garbage.root"
```

This is valid code: rule requester is calling somerule with (for example) `year="2017_MagnetUp"` and `polarity="garbage"`

Common mistakes: Wrong wildcard deduction

```
rule somerule:
    output: "afile_{year}_{polarity}.root"
    shell: "echo Running rule"

rule requester:
    input: "afile_2017_MagnetUp_garbage.root"
```

This is valid code: rule requester is calling somerule with (for example) `year="2017_MagnetUp"` and `polarity="garbage"`
This will eventually lead to an error →define what wildcard values are allowed

```
wildcard_constraints:
    year="201[5678]",
    polarity="Magnet(Up|Down)"
```

Note: these are regex strings

Exercises

Time to do it yourself!