Snakemake Hackathon

Materials for Snakemake hackathon organized on 22.05.24.

⚙ Prerequisites

- Depends on expected outcome; please read the supercomputing concept refresher and join the Snakemake intro session if you need it
- Supercomputer access

Snakemake introduction

Snakemake is a powerful workflow management system designed to create reproducible and scalable data analyses. Originating in the bioinformatics community, Snakemake has gained widespread use in various scientific and data-intensive fields. It allows users to define workflows in a simple, readable, and maintainable way using a Python-based language. With built-in support for parallel execution and resource management, Snakemake can efficiently handle workflows on various scales, from small-scale local computations to large-scale distributed computing environments.

Objectives

- Understand the components and features of a Snakefile
- Understand portability and reproducibility in the context of Snakemake
- Write a snakefile
- Run Snakemake from the shell
- Understand the key features of Snakemake

What is a workflow?

- Set of computational steps
- Step defined by input, process, output
- Connection through input and output files

_images/workflow.png

Workflow tools and why we need them?

Imagine having some data processing workflow that takes raw data, converts and filters it, and finally produces some result, for example a plot. Each step of the workflow is done using command line tools and scripts written in different languages. You could run everything manually: step by step, checking each output and passing it to the next script or tool.

Soon enough, you may start automating your work. The simplest way to automate would be to use bash script (if you use MacOS/Linux). Your script may look like this:

```
wget http://example.com/dataset_2024.zip
unzip dataset_2024.zip
python filter.py dataset_2024.txt filtered_dataset_2024.txt
python transform.py filtered_dataset_2024.txt transformed_dataset_2024.txt
Rscript stats.r filtered_dataset_2024.txt statistics.csv
Rscript plot.r statistics.csv plot.png
```

Although it works, this solution has some problems:

- How to quickly compute the results for a similar dataset from 2023?
- How to only run the `stats.r` script for unfiltered data?
- Have you noticed that I made a typo in the bash script and did not use transformed data for the `stats.r` script?
- How to compute the results for all datasets available (there are 100 of them) using a computer with 20 CPU cores?

As your workflows grow in terms of number of steps, or also the size of dataset, managing this workflow might become more and more problematic. Fortunately, over the years many solutions arose to manage and automate workflow execution. They offer plenty of features, supporting different stages of workflows: starting with the definition of a workflow, executing it and finally, gathering and sharing the results.

> By supporting the top layer, a workflow management system can promote the center layer, and thereby help to obtain true sustainability.

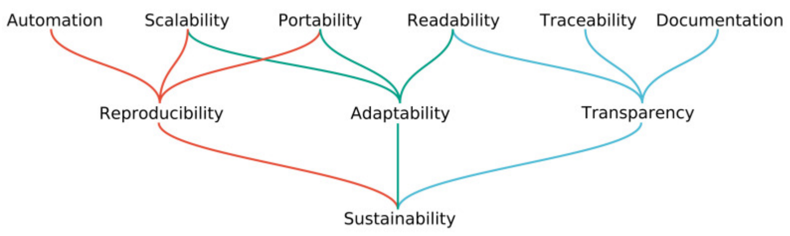


Image from Mölder F, Jablonski KP, Letcher B, Hall MB, Tomkins-Tinch CH, Sochat V, Forster J, Lee S, Twardziok SO, Kanitz A, Wilm A, Holtgrewe M, Rahmann S, Nahnsen S, Köster J. Sustainable data analysis with Snakemake. F1000Res. 2021 Apr 19;10:33. doi: 10.12688/f1000research.29032.2.

Why Snakemake?

- Highly popular tool: 11 new citations per week and over 1,000,000 downloads
- Available as Python package (pip, conda), but not Python specific!
- Python-based syntax (easy to read, quick to develop)
- Supports multiple different scripting languages and all command line tools
- Plenty of useful features for automating the work

It is important to remember that Snakemake does not require Python knowledge or limits you to using Python code.

Installation

To install Snakemake on your own computer, we can use Python Package Installer (`pip`). First, ensure you have Python installed on your system.

```
pip install snakemake
```

To verify the installation, run:

```
snakemake -v
```

You should be able to see the version of Snakemake installed on your system.

Snakefile

We can define Snakemake workflows using so-called `Snakefile`s. Workflows are defined in terms of rules. Each rule describes input files, outputs and a command or script to be run. Let’s have a look at a simple, one step workflow, that will copy the content of one file into another.

```
rule copy:
    input:
            "hello.txt"
    output:
            "hello_copy.txt"
    shell:
            "cp {input} {output}"
```

This `Snakefile` defines one input file `hello.txt`, one output file `hello_copy.txt` and a shell command `cp {input} {output}`. The shell command is formatted automatically by Snakemake. On runtime, `{input}` and `{output}` are substituted by the filenames provided.

To run this example, copy the above content into a file called `Snakefile` (without an extension) and create a `hello.txt` file in the same folder with some content using your favorite editor or in Linux/Mac you can also do:

```
echo "Hello Snakemake!" > hello.txt
```

To run the `Snakefile`, we run the following command:

```
snakemake --cores 1
```

Snakemake executes the `Snakefile` with the one rule in it. The content is copied! Note two things: the `--cores` flag and that we did not specified the `Snakefile`. By default, Snakemake will search for a `Snakefile` in the current directory. We can specify which `Snakefile` to run using the `--snakefile` flag. For example:

```
snakemake --cores 1 --snakefile Snakefile_2
```

will run the `Snakefile_2`. Thanks to that, now we can also have multiple `Snakefile`s in one directory. The `--cores` flag tells Snakemake how many resources (cores) from our computer Snakemake can use to execute the workflow. It is a mandatory flag, so one cannot omit it. You may however also use `--jobs` instead of `--cores` which does not make a difference on a local computer.

Rules

A rule do not necessarily have to have only one input or output. If so, we can use indices to refer to each input/output value:

```
rule sorter:
    input:
        "hello.txt",
        "hello2.txt"
    output:
        "sorted_greetings.txt",
    shell:
        "sort {input[0]} {input[1]} > {output}"
```

We can also use named inputs and outputs, and refer to them using their names:

```
rule sorter:
    input:
        a="hello.txt",
        b="hello2.txt"
    output:
        "sorted_greetings.txt",
    shell:
        "sort {input.a} {input.b} > {output}"
```

Wildcards

In the examples above we used hard-coded filenames to define the input and output values. This is not feasible for most workflows dealing with more than a few files. To generalize the workflows, we can use wildcards. Wildcards are variables that replace the actual filenames or any other value, like for example a path. Snakemake resolves them automatically based on either the target file or other input/outputs in the Snakefile. Let's add wildcards to the copy example, so it will work for any text file:

```
rule copy:
    input:
            "{data}.txt"
    output:
            "{data}_copy.txt"
    shell:
            "cp {input} {output}"
```

To run it, we have to tell Snakemake the value of the `data` wildcard. We can do that, by specifying what kind of output /target file we want to produce. In this case, we want to have `hello_copy.txt` - the copy of the `hello.txt` file.

```
snakemake --cores 1 hello_copy.txt
```

Snakemake parses the `hello_copy.txt` argument and detects that the wildcard is equal to `hello`. Then, Snakemake looks for the `hello.txt` file.

Python code in Snakefile

Another powerful feature is the possibility to add Python code and function inside of Snakefile. We also have access to Python libraries, by using traditional `import` keyword. For example, the following code searches for all text files in the given directory, and appends their content in the `copy.txt` file.

```
import os

def find_txt_files(path):
    txt_files = [file for file in os.listdir(path) if file.endswith(".txt")]
    return txt_files

rule append:
    input:
            find_txt_files(".")  # . means the current working directory
    output:
            "results/copy.txt"
    shell:
            "cat {input} >> {output}"
```

How is this different than wildcards? Here, we did not resolve or specify any filename. We can call `snakemake --cores 1` without any arguments. This workflow searches for the input files by itself, without any guidance. Use with care, and for example test that the files that are found are the ones you actually want to process by doing a dry-run, using `--dry-run` when calling Snakemake.

Another way of using Python is replacing the shell command with Python script:

```
rule copy:
    input:
            "hello.txt"
    output:
        "hello_copy.txt"
    run:
        with open(input[0]) as in_f, open(output[0], "w") as out_f:
            for l in in_f:
                out_f.write(l)
```

This approach is fine for small code snippets. For bigger scripts, it is better to move code to a separate file and call it in the `Snakefile`. Let's create the `copy.py` file:

```
with open(snakemake.input[0]) as in_f, open(snakemake.output[0], "w") as out_f:
    for l in in_f:
        out_f.write(l)
```

and modify the `Snakefile`:

```
rule copy:
    input:
        "hello.txt"
    output:
        "hello_copy.txt"
    script:
        "copy.py"
```

In the Python file, we have access to the `snakemake` object, which allows us to get access to the content of the `input` and `output` directives! We do not need to parse any command line arguments. The access is granted by the `script` keyword, which works like a wrapper around the script. Similar integrations are available for other languages as well: R, Markdown, Julia, Rust, Bash and Jupyter Notebook. See the current list in the Snakemake documentation.

Dependencies between rules

So far, we only examined `Snakefile`s with only one rule. Let's have a look at a bigger workflow.

```
rule concatenate:
    input:
        expand("data/file{n}.txt", n=[1, 2, 3])
    output:
        "results/concatenated.txt"
    shell:
        "cat {input} > {output}"

rule count_words:
    input:
        "results/concatenated.txt"
    output:
        "results/word_count.txt"
    shell:
        "wc -w {input} > {output}"
```

This workflow concatenates three input files (`data/file1.txt`, `data/file2.txt`, `data/file3.txt` using `expand`, another way to define e.g. filenames more specific than wildcards) and then counts how many words there are in the files using the command line tool `wc`. The `Snakefile` consists of three rules. The dependencies between rules are set by the input and output filenames. The rule `concatenate` generates the output file: `results/concatenated.txt`, which then is used as an input to the `count_words` rule.

Snakemake uses top-down approach to automatically resolve the dependencies between rules. That means, Snakemake will start from the last rule (so-called `target rule`) and go backwards, searching for the input needed to execute that step. By default, the target rule is the first rule in the Snakefile. In this case, we want to run the `count_words` rule as the target (first we concatenate, then we count the words). We have to somehow tell Snakemake what is the final output of this workflow. To achieve that, a common practice is adding a mock `rule all` at the beginning of the `Snakefile` that encapsulates the final outputs of the workflow:

```
rule all:
    input:
        "results/word_count.txt"

rule concatenate:
    input:
        expand("data/file{n}.txt", n=[1, 2, 3])
    output:
        "results/concatenated.txt"
    shell:
        "cat {input} > {output}"

rule count_words:
    input:
        "results/concatenated.txt"
    output:
        "results/word_count.txt"
    shell:
        "wc -w {input} > {output}"
```

Snakemake will see that we want the `results/word_count.txt` file, and will search in the `Snakefile` to see what rule produces this exact output. Then, it will repeat this process recursively until all dependencies are resolved. By doing so, a so-called directed acyclic graph (DAG) is created. It is a step-by-step execution plan for Snakemake. We can see and analyze this graph by calling Snakemake with `--dag` flag:

```
snakemake --cores 1 --dag
```

The result can be visualized using either the `GraphViz` package (which needs to be installed separately, see example call below), or using GraphVizOnline.

```
snakemake --cores 1 --dag | dot -Tpng > dag.png
```

Another way to link rules in the `Snakefile` would be to directly refer to the outputs of the rules by using the rule's name. This approach is more robust when the execution order of the steps matters, as we can freely adjust output names, without breaking the workflow. Remember that you can only refer to the rules that were defined before, so the `all` rule cannot be modified that way!

```
rule all:
    input:
        "results/word_count.txt"

rule concatenate:
    input:
        expand("data/file{n}.txt", n=[1, 2, 3])
    output:
        "results/concatenated.txt"
    shell:
        "cat {input} > {output}"

rule count_words:
    input:
        rules.concatenate.output
    output:
        "results/word_count.txt"
    shell:
        "wc -w {input} > {output}"
```

Force execution

Snakemake is designed to efficiently manage the execution of complex workflows by only computing the parts that are necessary. Specifically, it checks the presence and timestamps from the metadata of the output files specified in your workflow rules. If an output file already exists and is up to date, Snakemake will skip the computation steps that produce that file, saving time and resources by avoiding redundant computations.

However, there are situations where you might want to recompute the entire workflow, regardless of the existing files. In such cases, you can force Snakemake to recompute all the steps by using the `-F` or `--forceall` flag. This tells Snakemake to ignore the existing output files and re-run all the rules as if the files were missing.

```
snakemake --cores 1 --forceall
```

Besides forced execution, Snakemake may execute a rule again in two more cases: if the rule definition has changed since the last execution (for example by adding a new output file) or if the script that is executed has changed (only when using the `script` keyword).

Parallelization

So far, we only used Snakemake with one core (`--cores 1`). Let's say we have a workflow that processes four files. Each file is processed with two steps: `modify_file` and `count_words`.

The rule `modify_file` copies the content of the input file to the outputfile using `cat` and redirection `>`, then appends the content of the params directive to the output file using `echo` and redirect append `>>`. In the end the process pauses for 5 seconds using `sleep 5`.

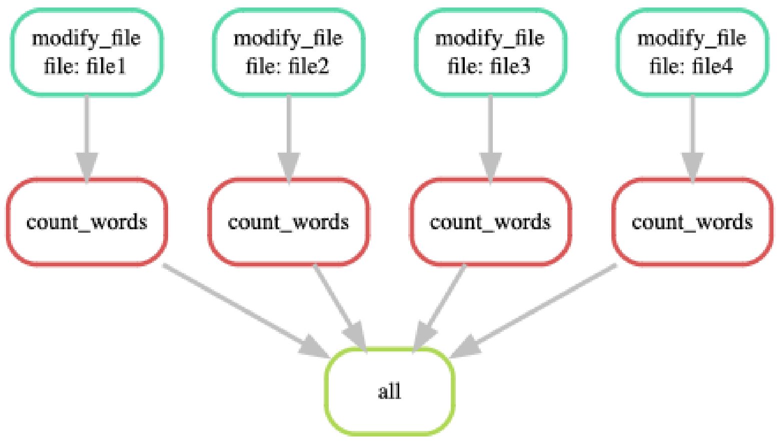
The rule `count_words` uses the command line tool `wc` with the argument `-w` to count the number of words in the input file and writes the number to the output file. In the end the process pauses for 5 seconds using `sleep 5`.

```
rule all:
    input:
        "results/word_count_file1.txt",
        "results/word_count_file2.txt",
        "results/word_count_file3.txt",
        "results/word_count_file4.txt",

rule modify_file:
    params:
        msg="This was modified by Snakemake!"
    input:
        "data/{file}.txt"
    output:
        "results/modified_{file}.txt",
    shell:
        "cat {input} > {output} && echo '{params.msg}' >> {output}  && sleep 5"

rule count_words:
    input:
        "results/modified_{file}.txt",
    output:
        "results/word_count_{file}.txt",
    shell:
        "wc -w {input} > {output} && sleep 5"
```

When we create a DAG for that workflow, we see that there are four branches in the graph - one for each file.



If we run Snakemake with the following command:

```
snakemake --cores 1
```

The execution will take a little bit more than 40 seconds. Snakemake uses only one core, executing one job at a time. Let’s increase number of cores to 2. To run the full workflow again, even though the files have already been processed, remember to add the `--forceall` flag.

```
snakemake --cores 2 --forceall
```

Now we can see in the terminal output, that Snakemake is running two jobs at a time, using two cores. The execution time was around two times faster. Snakemake automatically detected, that some parts of the workflow can be run in parallel, and used the provided resources to parallelize the work. In this case, each file can be processed independently, there is no aggregation or combining the results.

Error recovery and re-entry

Snakemake offers error recovery features. Analyzing the above example, what would happen if one of the input files that we try to modify is corrupted, i.e. not readable? Snakemake will automatically stop the execution of the entire workflow, even though three other files can be processed correctly. Although this behaviour may be useful in some cases, sometimes we may want to compute as many results as we can. To enable this, we can use the `--keep-going` flag. It makes Snakemake keep going as far into workflow as it possibly can, computing all results that do not rely on corrupted files or failed jobs.

```
snakemake --cores 2 --keep-going
```

Another error recovery strategy is retrying, which is useful for example when your workflow relies on some online resource.

```
rule get_data_from_server:
    output:
        "test.txt"
    retries: 3
    shell:
        "curl https://some.unreliable.server/test.txt > {output}"
```

In cases where a server does not respond with any data, the `curl` command exists with an error. Snakemake quickly retries the job three times. Retrying can be set globally for all jobs using the `--retries` flag followed by the number of retries.

Reporting

Snakemake generates execution logs with the content that is also printed in the terminal and stores them in the (hidden) `.snakemake` directory located in the directory where the `snakemake` command is executed . After executing a workflow, Snakemake saves metadata about the execution in that directory. To generate a proper report, we use the `--report` flag when executing the `snakemake` command followed by the filename for the report.

Reports in Snakemake are `html` documents that are generated after workflow execution. They summarize the work done by Snakemake together with some statistics. To generate `html` reports, an extra Python package called `pygments` needs to be installed, for example using `pip`:

```
pip install pygments
```

Let's see how we can generate a report. For that, we can use one of the previous workflows.

```
rule all:
    input:
        "results/word_count.txt"

rule concatenate:
    input:
        expand("data/file{n}.txt", n=[1, 2, 3])
    output:
        "results/concatenated.txt"
    shell:
        "cat {input} > {output}"

rule count_words:
    input:
        "results/concatenated.txt"
    output:
        "results/word_count.txt"
    shell:
        "wc -w {input} > {output}"
```

```
snakemake --report report.html
```

We can add more information to the report, including the output data of our workflow. In such cases, data is added to the HTML document and can be downloaded. This way we can share the results (at least if they are quite small) with the report and execution information!

```
rule all:
    input:
        "results/word_count.txt"

rule concatenate:
    input:
        expand("data/file{n}.txt", n=[1, 2, 3])
    output:
        report(
            "results/concatenated.txt",
            category="Step 1"
        )
    shell:
        "cat {input} > {output}"

rule count_words:
    input:
        "results/concatenated.txt"
    output:
        report(
            "results/word_count.txt",
            category="Step 2"
        )
    shell:
        "wc -w {input} > {output}"
```

By adding the `report()` around our output files, we include both output text files in the report. When opening the report in a web browser, we can now also access the output files and download them.

Monitoring

> Warning: this section may not work for Windows users!

Snakemake also offers an option for monitoring the workflow from outside of the current workflow execution using an extra tool called `panoptes`. It provides a simple web page with an API to which Snakemake connects. For Linux and Mac, `panoptes` can be installed using `pip`:

```
pip install panoptes-ui
```

Panoptes will run a server on `localhost` (i.e. on our computer), which we can start using the following command:

```
panoptes
```

To close the server, we can use `CTRL+C` in the terminal. The server is running by default on `http://127.0.0.1:5000`. If we open our browser and visit this website, we will see the graphical interface for monitoring workflows. Note that now we cannot use our terminal, because it is used by `panoptes`. We have to open a new one to be able to run new commands. To connect Snakemake to the monitoring, we modify our `snakemake` command:

```
snakemake --cores 1 --wms-monitor http://127.0.0.1:5000
```

If you server has a different URL, replace it in the command above. When we execute the workflow, we can see some basic statistics about the execution.

Warning: this set-up might be not secure enough for the production use. See this thread for more details.

Workflow reproducibility

Reproducibility in computational workflows ensures that an analysis can be consistently repeated with the same results, which is crucial for validating findings, ensuring transparency, and facilitating collaboration. It allows other researchers to verify results, understand methodologies, and build upon previous work.

Project structure

Snakemake provides recommendations regarding project structure.

```
├── .gitignore
├── README.md
├── LICENSE.md
├── workflow
│   ├── rules
│   │   ├── module1.smk
│   │   └── module2.smk
│   ├── envs
│   │   ├── tool1.yaml
│   │   └── tool2.yaml
│   ├── scripts
│   │   ├── script1.py
│   │   └── script2.R
│   ├── notebooks
│   │   ├── notebook1.py.ipynb
│   │   └── notebook2.r.ipynb
│   ├── report
│   │   ├── plot1.rst
│   │   └── plot2.rst
│   └── Snakefile
├── config
│   ├── config.yaml
│   └── some-sheet.tsv
├── results
└── resources
```

Following the documentation:

> The workflow code goes into a subdirectory called `workflow`, while the configuration is stored in a subdirectory called `config`. Inside of the `workflow` subdirectory, the central `Snakefile` marks the entrypoint of the workflow (it will be automatically discovered when running snakemake from the root of above structure).
>
> Workflows that are set up in above structure can be more easily re-used and combined via the Snakemake module system. Such deployment can even be automated via Snakedeploy. Moreover, by publishing a workflow on Github and following a set of additional rules the workflow will be automatically included in the Snakemake workflow catalog, thereby easing discovery and even automating its usage documentation.

Containerization

Containers, like Docker or Singularity/Apptainer, provide a lightweight and portable way to package and run applications. They encapsulate an application and its dependencies, ensuring it runs consistently across different environments. Unlike virtual machines, containers share the host system's kernel but isolate the application's processes, filesystem, and resources. This makes containers more efficient in terms of performance and resource usage, allowing for quick startup times and easier scalability.

To use containers with Snakemake, we add the `container` keyword inside a rule, linking either to a local or external container image:

```
rule copy:
    container:
        "docker://alpine:3.14"
    input:
            "hello.txt"
    output:
            "hello_copy.txt"
    shell:
            "cp {input} {output}"
```

To execute that Snakefile, we have to add the `--software-deployment-method` flag:

```
snakemake --cores 1 --software-deployment-method apptainer
```

When executed, Snakemake will run the `copy` job inside the `Alpine` (one of the Linux operating systems) image. Note that running that code requires the `apptainer` command to be available on your system.

Summary

Presented features are only a subset of all things that Snakemake has to offer. Feel free to check the official documentation for the latest updates! There are also tutorials that cover most of the basics.

Snakemake offers plenty of features that automate the process of creating, running and overseeing the execution. You can document your workflow using DAG and reports. It is easier to make better usage of resources thanks to the implicit parallelization. Error recovery helps with dealing with errors, which are inevitable.

So far, we only used a personal computer. In that mode, Snakemake is using a so-called `local executor`, which uses resources of our computer. After a short break, you will learn how to use other types of executors, that allow for integrating with supercomputers and leveraging resources that they offer.

Snakemake on the supercomputer

Objectives

- TBD

Running Snakemake on CSC's Puhti or LUMI: Slides.

Prerequisite: Portable workflow

Check your filepaths!

Working with modules

`module load snakemake/version`

Snakemake cluster execution

Snakemake provides a generic cluster executor as plugin, which can be used to run Snakemake workflows on clusters:

```
module load snakemake

snakemake -s Snakefile --jobs 1 \
 --latency-wait 60 \
 --executor cluster-generic \
 --cluster-generic-submit-cmd "sbatch --time 10 \
 --account=project_xxxx --job-name=hello-world \
 --tasks-per-node=1 --cpus-per-task=1 --mem-per-cpu=4000 --partition=test"
```

Submit Snakemake as a Batch Job; request more resources if needed. All rules are run in the same job allocation.

```
#!/bin/bash
#SBATCH --job-name=myTest
#SBATCH --account=project_xxxxx
#SBATCH --time=00:10:00
#SBATCH --mem-per-cpu=2G
#SBATCH --partition=test
#SBATCH --cpus-per-task=4

module load snakemake/8.4.6
snakemake -s Snakefile --use-singularity --jobs 4
```

Another way would be to use slurm executor plugin or HyperQueue: `snakemake --executor cluster-generic --cluster-generic-submit-cmd "hq submit ..."`

Good practices

- Use version control for your workflows
- Use containers for portability
- Avoid unnecessary read and write operations
- Summarize small jobs/job steps into one job
- Use restarting option when running long jobs
- Avoid creating a lot of files, especially in the same folder
- Remove temporary files after the job is finnished
- Separate serial from parallel jobs for efficient use of resources

Keypoints

- TBD

Toy example: Snakemake at Scale

Objectives

- TBD

This example is adapted from "Using CSC Computing Environment Efficiently" course.

Use Containers as Runtime Environment

One can use Singularity/Apptainer container as an alternative to native installations for better portability and reproducibility. If you don't have a ready-made container image for your needs, you can build a Singularity/Apptainer image using **–fakeroot** option.

For the purpose of this tutorial a pre-built container image which has all the software stack needed is provided to run snakemake workflow at scale.

Use HyperQueue Executor to Submit Jobs

If the workflow manager is using `sbatch` for each process execution (i.e., a rule), and you have many short processes, it's advisable to switch to HyperQueue to improve throughput and decrease load on the system batch scheduler.

You can load HyperQueue and Snakemake modules on different clusters as described below:

CSC Puhti   LUMI

```
module load hyperqueue/0.16.0
module load snakemake/8.4.6
```

One can use HyperQueue executor settings depending on the Snakemake version as below:

version 8.x.x   version 7.x.x

`snakemake --executor cluster-generic --cluster-generic-submit-cmd "hq submit ..."`

Submit Snakemake Workflow on Cluster

Download tutorial materials (scripts and data), which have been adapted from the official Snakemake documentation, from CSC Allas object storage as below:

```
wget https://a3s.fi/snakemake_scale/snakemake_scaling.tar.gz
tar -xavf snakemake_scaling.tar.gz
```

The downloaded material includes scripts and data to run snakemake pipeline. You can use `snakemake_hq_puhti.sh` whose content is posted below:

```bash
#!/bin/bash
#SBATCH --job-name=snakemake
#SBATCH --account=<project>  # replace <project> with your project, e.g. for CSC:
project_2001234
#SBATCH --partition=small
#SBATCH --time=00:10:00
#SBATCH --nodes=1
#SBATCH --ntasks-per-node=1
#SBATCH --cpus-per-task=40
#SBATCH --mem-per-cpu=2G

module load hyperqueue/0.16.0
module load snakemake/8.4.6

# Specify a location for the HyperQueue server
export HQ_SERVER_DIR=${PWD}/hq-server-${SLURM_JOB_ID}
mkdir -p "${HQ_SERVER_DIR}"

# Start the server in the background (&) and wait until it has started
hq server start &
until hq job list &>/dev/null ; do sleep 1 ; done

# Start the workers in the background and wait for them to start
srun --exact --cpu-bind=none --mpi=none hq worker start --cpus=${SLURM_CPUS_PER_TASK} &

hq worker wait "${SLURM_NTASKS}"

snakemake -s Snakefile --jobs 1 --use-singularity --executor cluster-generic --cluster-
generic-submit-cmd "hq submit --cpus 5"

# for snakemake versions 7.x.xx, use command: snakemake -s Snakefile --jobs 1 --use-
singularity --cluster "hq submit --cpus 2"

# Wait for all jobs to finish, then shut down the workers and server
hq job wait all
hq worker stop all
hq server stop
```

How do you parallelise snakemake workflow jobs?

The default script provided above is not optimised to run in high-throughput way as snakemake workflow manager just submits one job at a time to the hyperqueue scheduler. You can parallelise workflow tasks (i.e., rules in snakemake) by submitting more jobs from *snakemake* command as below:

```bash
snakemake -s Snakefile --jobs 8 --use-singularity --executor cluster-generic --cluster-
generic-submit-cmd "hq submit --cpus 5"
```

You can correct above modification in the batch script (and use your own project number in sbatch directives) before submitting the Snakemake workflow job to the HPC cluster as below:

```
sbatch snakemake_hq_puhti.sh
```

One can also use more than one node to achieve even more high-throughput as HyperQueue can make use of multi-node resource allocations.

Please note that just by increasing the number jobs will not alone automatically run all those jobs. *Jobs* parameter from *snakemake* is just a maximum limit for concurrent jobs. Jobs will be eventually run when resources are available. In our case we submitted 8 parallel jobs, each taking 5 CPUs as we reserved 40 CPUs in batch script. In practice it is a good idea to dedicate few CPUs for workflow manager itself.

Follow the progress of jobs

You can already check the progress of your job by simply observing the current folder where you can see lot of new task-specific folders are being created. However, there are formal ways to check the progress of your jobs as shown below:

Full job monitoring

```
squeue -j <slurmjobid>
# or
squeue --me
# or
squeue -u $USER
```

~~~
````{group-tab} Sub task monitoring
  ```bash
  module load hyperqueue
  export HQ_SERVER_DIR=$PWD/hq-server-<slurmjobid>
  hq worker list
  hq job list
  hq job info <hqjobid>
  hq job progress <hqjobid>
  hq task list <hqjobid>
  hq task info <hqjobid> <hqtaskid>
~~~

How do you clean different task-specific folders automatically?

HyperQueue creates task-specific folders (i.e., job-`<n>`) in the same directory from where you have submitted batch script. These are sometimes useful for debugging. However if your code is working fine, the creation of such large number folders may be annoying besides causing some overhead to parallel file systems like Lustre. You can prevent creating such task-specific folders by setting `stdout` and `stderr` flags to `none` as shown below:

```
snakemake -s Snakefile -j 24 --use-singularity --executor cluster-generic --cluster-generic-submit-cmd "hq submit --stdout=none --stderr=none --cpus 5 "
```

More Information

- CSC documentation on Snakemake
- Snakemake official documentation

Keypoints

- TBD ``

About the course

Snakemake is a common entry tool in the world of computational workflows, especially in Bioinformatics.

The tool is designed to support you in setting up a workflow system, which opposed to a “your favorite language here” script, helps you keep track of your processes.

For Snakemake, the user has to define inputs and outputs and the tool will figure out the order of steps and which steps can be run in parallel.

This Hackathon is open for all, free of charge and independent of field of Science and it is all about getting your own workflows to the supercomputer (Puhti, Mahti, Lumi or other).

Content

We will offer short introduction to snakemake and a supercomputing concepts refresher session in the beginning, which you can attend if needed.

Then we will talk about opportunities and challenges with moving workflows from laptop to the supercomputer and give tips on how to go about it.

The afternoon is reserved for your own work. You can either work on your own “Snakefile”, moving your workflow from laptop to supercomputer or work on our toy example, to help understand the concepts better before moving to your own work.

Learning outcomes

After this course, participants will be able to…

- Understand what is snakemake
- Understand when it does make sense to use Snakemake, when not
- Implement own workflow with Snakemake
- Move own workflow to the supercomputer

See also

- CSC bio workflows with Nextflow course materials
- CSC workflows on HPC considerations - slides
- CSC High Throughput Computing docs page
- Uppsala snakemake BYOC
- Snakemake tutorial
- CodeRefinery lesson
- Carpentries lesson
- Snakemake workflows catalog
- CSC Snakemake tutorial
- MultiXScale Workflows material
- Universe HPC Snakemake lesson