

Snakemake Hackathon

Materials for [Snakemake hackathon](#) organized on 22.05.24. Licensed under CC-BY-4.0 license.



⚙️ Prerequisites

- Depends on expected outcome; If needed, please read this material as supercomputing concept refresher: https://coderefinery.github.io/TTT4HPC_resource_management/# (If you are interested, also check the rest of the materials from “Tuesdays Tools and Techniques for HPC” (it is quite useful for daily working on clusters): <https://scicomp.aalto.fi/training/scip/ttt4hpc-2024/>)
- Supercomputer access

Snakemake introduction

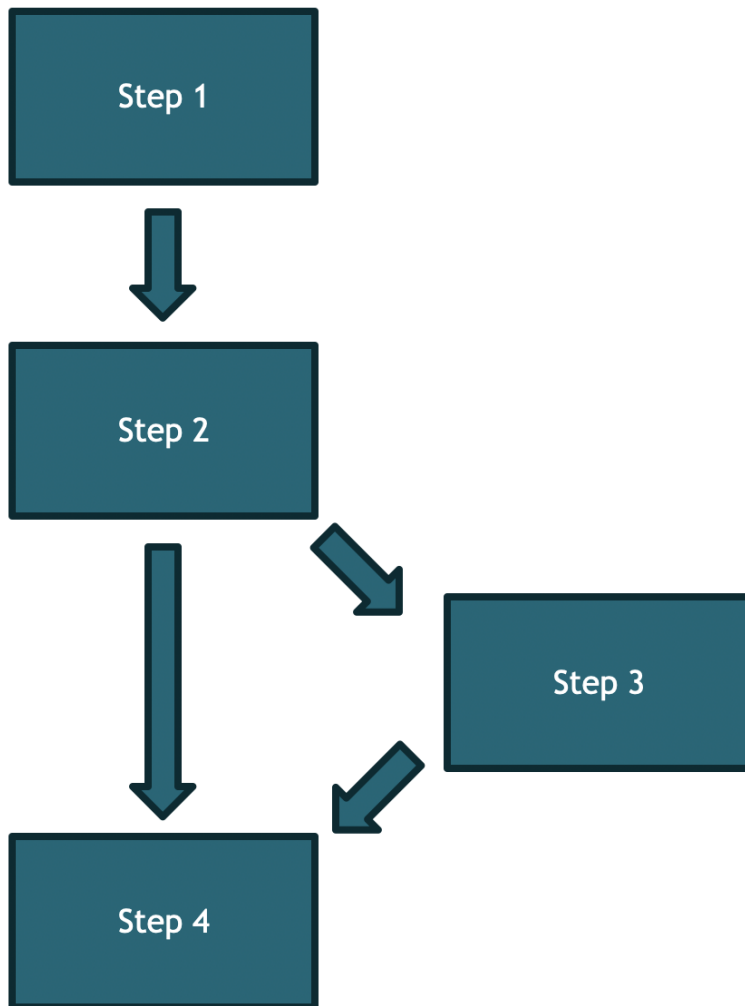
Snakemake is a powerful workflow management system designed to create reproducible and scalable data analyses. Originating in the bioinformatics community, Snakemake has gained widespread use in various scientific and data-intensive fields. It allows users to define workflows in a simple, readable, and 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



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.

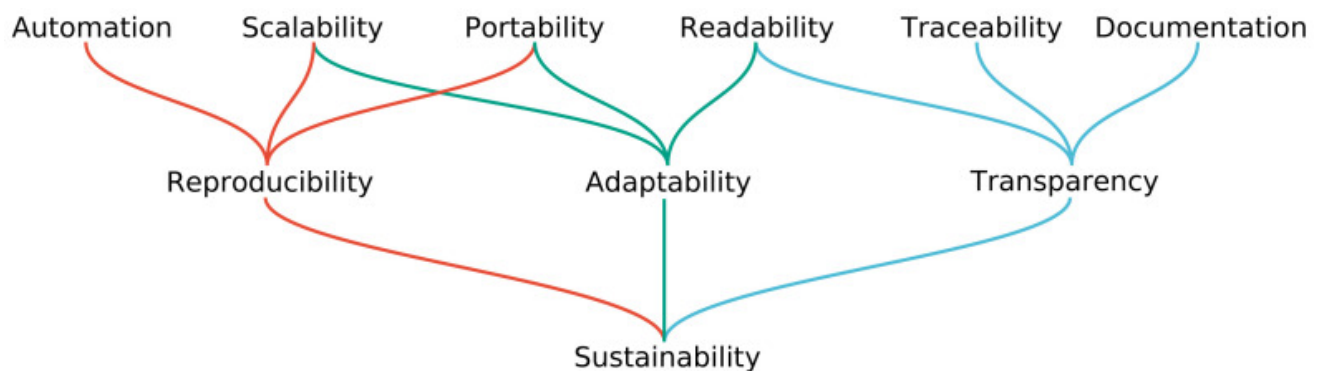


Image from Mölder F, Jablonski KP, Letcher B, Hall MB, Tomkins-Tinch CH, Sochat V, Forster J, Lee S, Twardziok SO, Kanitz A, Wilm A, Holtgrewe M, Rahmann S, Nahnsen S, Köster J. Sustainable data analysis with Snakemake. *F1000Res*. 2021 Apr 19;10:33. doi: 10.12688/f1000research.29032.2.

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 specify 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 does 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`.

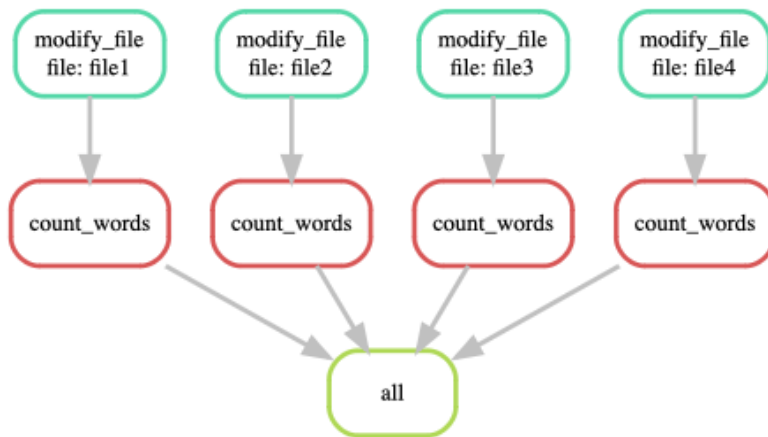
The rule `count_words` uses the command line tool `wc` with the argument `-w` to count the number of words in the input file and writes the number to the output file. In the end the process pauses for 5 seconds using `sleep 5`.

```
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 your 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:

```
shell.executable("/bin/sh")

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

Note that in this case we also have to define the shell executable available within the alpine container with `shell.executable("/bin/sh")`.

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

- Understand the different ways Snakemake can be run on clusters
- Evaluate which way suits your needs
- Get an idea on how to make own workflow ready for the supercomputer

Note on versions: These materials assume Snakemake version 8.0 or later. Earlier (before version 8.0) Snakemake versions deal differently with SLURM. Please check the documentation for your version of snakemake.

Cluster Snakemake installation

1. Preinstalled and made available via module: `module load snakemake/version`
2. Own Conda/Pip based installation
3. Own container based installation

Always check first if your cluster has the tool available as **module**, then you do not have to worry about installation yourself. If you need additional packages, you can usually add own packages by `pip install` for your own user or project. It may also be possible to load multiple modules at the same time (feasibility depends on the modules). If that for some reason does not work, you can create your own installation with Conda or Pip, or by making use of containers. **Check your clusters documentation for information on how to do these.** Ask your friendly cluster support for help or advice, if needed.

Ways of running Snakemake on clusters

Within one SLURM job

Interactive

You can run snakemake the same way you would run it on your own computer also on cluster within one interactive or normal batch job (one resource allocation for the whole workflow (= all rules)). The interactive session is a useful first step to try it out, but does not scale well (only to the resource limits of one interactive job).

In an interactive job (using the interactive partition; some clusters support starting an interactive session with the `sinteractive` command), you would load the snakemake module, and then run `snakemake --cores 1` (or more, if you reserved more) as you would on your local machine.

You can run snakemake in the background for example with `screen`.

Sbatch script

Another way to run snakemake in **one job** is to use an sbatch script for job submission to run the workflow non-interactive.

```
#!/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/version
snakemake --cores 4
```

Note how 4 CPUs are reserved (`#SBATCH --cpus-per-task=4`) and made available to the workflow (`--cores 4`). Within the Snakefile resource distribution and parallelization can be controlled using `threads:X` directive, see the [Snakemake documentation on threads](#).

Slurm integration with the [Snakemake SLURM executor plugin](#)

With the plugin you can define SLURM batch job resources for your snakemake workflow instead of writing an sbatch file.

Snakemake provides a [Snakemake SLURM executor](#) as plugin, which can be used to let Snakemake run workflows on clusters. For this, you would start an interactive session, from which you could call :

```
module load snakemake/version

snakemake --jobs 1 --executor slurm --default-resources slurm_account=project_xxx
slurm_partition=test runtime=10 mem_mb_per_cpu=2000 cpus_per_task=4
```

This command would submit **one new job** (defined by `--jobs 1`) to SLURM for running the whole workflow in that job, similar to the sbatch script above. Instead of defining `cpus_per_task` for the whole workflow, one could also define `threads:X` for each rule.

Other SLURM resource specifications can be found from the table in the [SLURM executor plugin documentation](#).

The SLURM integration also provides the possibility to start multiple SLURM jobs (specified by setting `--jobs` to `> 1`) for you; for example one for each step. However, with this option you would have to wait in queue for every single step to be executed which generates a lot

of overhead, especially for short job steps. Using the Snakemake SLURM integration this way should therefore be avoided in cases where single steps are short. A better solution to scale up your Snakemake workflow (also to multiple nodes) is to use Hyperqueue instead:

Snakemake at scale using Hyperqueue

[HyperQueue \(HQ\)](#) is an efficient sub-node task scheduler. Instead of submitting each of your computational tasks as separate Slurm jobs or job steps, you can allocate a large resource block and then use HyperQueue to submit your tasks to this allocation, even when running your workflow on multiple nodes. The [generic cluster executor](#) plugin can be used to configure hyperqueue runs, e.g.

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

- Useful for high-throughput jobs (many steps and rules on many files, needing a lot of resources)
- Extra benefit: Multiple nodes can be used under the same job allocation

For more information, please check out the provided Hyperqueue example.

Config/Profile files

Snakemake takes many parameters as command line arguments/flags. However, when working with clusters these can become quite many very fast. Many of which also do not change very often, e.g. `slurm-account`. One way to make the command line call shorter, is to define some of the parameters in [profiles](#). These can be global or workflow specific.

A profile is written in `YAML format` and usually called `config.yml` and can be provided to Snakemake using the `--profile` flag (Note to only provide the path where Snakemake should look for the profile file, e.g. `profiles/slurm/` (without the profile filename)). An option defined by `--someoption` in the Snakemake command line call becomes `someoption:` in the profile. Example profile:

```
jobs: 1
default-resources:
  slurm_account: project_xxx
```

Check out another [Snakemake profile example from the Tuesdays Tools and Techniques course](#).

Good practices

- General:
 - As with all code, it is advised to also keep your Snakefiles under **version control**, e.g. using `git`
 - Consider **containerizing** your whole workflow for portability

- When running Snakemake on clusters:
 - Try to **avoid unnecessary read and write operations**, check if use of fast local disk is possible
 - Consider summarizing small jobs/job steps into one job when you want to use the SLURM plugin to limit SLURM accounting overhead, e.g. by [job grouping](#)
 - Try to avoid creating a lot of files, especially in the same folder, [Snakemake can remove temporary files after the job is finished](#).

Cluster specific information

CSC Puhti or LUMI

[Slides on running Snakemake on CSC's Puhti or LUMI](#).

Keypoints

- Use the `snakemake module` if your cluster provides one; add own packages or modules where necessary
- If that fails, consider creating your own environment; follow cluster specific instructions
- Use interactive jobs to test and setup your Snakemake workflow
- Use normal batch jobs for running your tested workflows via sbatch script or using the Snakemake SLURM executor (`--jobs = 1`)
- Use Hyperqueue to scale your workflow on the cluster, especially when you need multiple nodes

Hackathon

This time is reserved for you. You can start implementing your own workflow with Snakemake, move your existing workflow to a cluster or follow any of our toy examples, based on interest.

Snakemake in general and on cluster

If you have not used Snakemake before, we recommend to start on your laptop and follow the [CodeRefinery Snakemake demo](#) instructions to create your first workflow and run it on your laptop (Installation: [Snakemake installation](#)) or within an interactive job on your cluster (resource needs are minimal, 2GB of memory and one core should more than suffice).

You can also use the CodeRefinery example as a base and try and run it on a cluster as batch job or with Hyperqueue instead.

Using Hyperqueue

If you want to try out Hyperqueue, we suggest to follow the [Hyperqueue example](#).

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
- Decide if using Snakemake fits their workflow
- 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](#)