Introduction to High-Throughput-Computing: SnakeMake

Luka Leskovec, UL FMF

4. 7. 2023

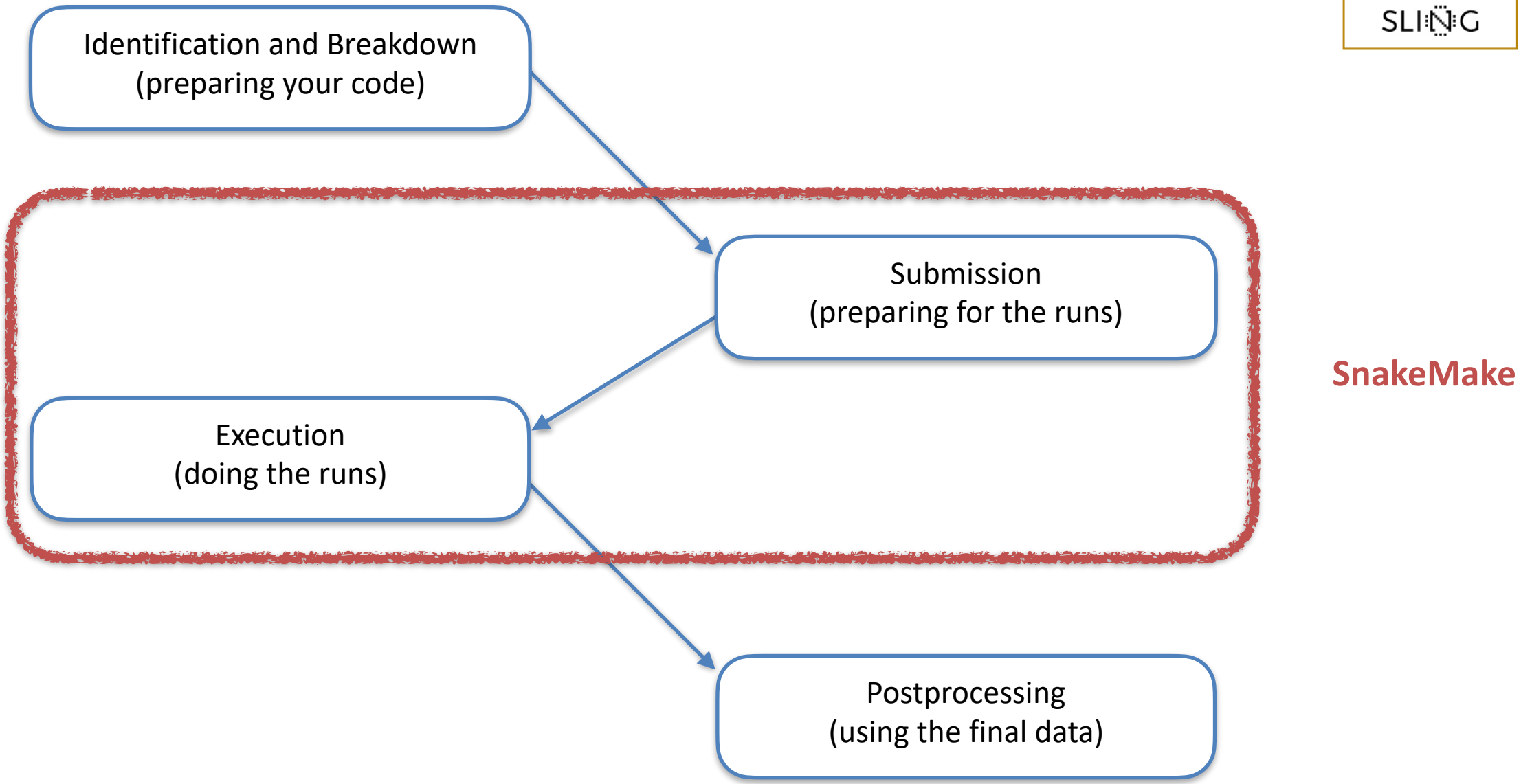
Types of SuperComputing

- HPC - high-performance-computing
  (one process distributed over large amount of cores)
- HTC - high-throughput-computing
  (large number of processes running on a large amount of cores)
- Grid computing
  (sharing of smaller size clusters amongst geographically disperse locations)
- Cloud computing (*)
- Quantum computing (**)

High-Throughput-Computing (HTC)

- (mostly) independent processes, little communication
- each running on 1 (or a few) cores
- individual tasks do not require extensive computational resources
- large number of tasks
- massive amounts of data
- **exploiting (simple) parallelism and distributing the workload across multiple computing resources**
- scientific research, data analysis, simulations, …

High-Throughput-Computing



Example I: random numbers

PROBLEM:

a) Write a python program, that takes 3s to generate a random number between 0 and 99.

b) generate 10 random numbers

Solution I: random numbers

random_print.py

```python
import time
import random


for i in range(10):
    time.sleep(3)
    random_number = random.randint(0, 99)
    print("result:", random_number)
```

simple solution, takes 30s, uses a single core

```
time python3 ./random_print.py
result: 99
result: 92
result: 26
result: 69
result: 57
result: 56
result: 3
result: 28
result: 90
result: 60
python3 ./random_print.py  0.02s user 0.02s system 0% cpu 30.106 total
```

Solution II: random numbers

app.py

```python
import time
import random

time.sleep(3)

random_number = random.randint(0, 99)
print("result:", random_number)
```

script.sh

```
#!/bin/zsh

for i in {1..10};
do
    python3 ../app.py
done
```

```
./script.sh
result: 47
result: 27
result: 27
result: 74
result: 82
result: 20
result: 12
result: 7
result: 42
result: 42
./script.sh  0.20s user 0.08s system 0% cpu 30.384 total
```

Solution III: random numbers

app.py

```python
import time
import random

time.sleep(3)

random_number = random.randint(0, 99)
print("result:", random_number)
```

Snakefile

```
results = "output_{i}.txt"

all_results = expand(results, i = [x for x in range(10)] )

rule all:
  input:
    all_results

rule run_app:
  input:
  output:
        "output_{i}.txt"
  shell:
        "python3 ../app.py  > {output}"
```

```
Building DAG of jobs...
Using shell: /bin/bash
Provided cores: 10
Rules claiming more threads will be scaled down.
Job stats:
job         count    min threads    max threads
-------  -------  -------------  -------------
all             1              1              1
run_app        10              1              1
total          11              1              1
```

...

```
snakemake -c 10  0.37s user 0.20s system 16% cpu 3.542 total
result: 24
result: 67
result: 31
result: 5
result: 36
result: 38
result: 11
result: 0
result: 40
result: 99
```

Solutions: overview

EURO SLING

solution I

- simple python code
- ran directly from terminal
- runtime: 30s
- parallelisation potential:
  - threads in python
  - mpi4py
  - trivial parallelisation

solution II

- simple python code
- ran within a script
- runtime: 30s

solution III

- simple python code
- Snakemake
- runtime: 3s

The basics of Snakemake

paralelizing something simple

what is Snakemake

- a python variant of Make
- make rules + python = snakemake
- can define a workflow, i.e. do one thing, then another etc.
- dependencies between rules are implicit through input/output parts of rules
- can run shell code or python code

```
rule NAME:
    input:
        "path/to/inputfile", "path/to/other/inputfile"
    output:
        "path/to/outputfile", "path/to/another/outputfile"
    shell:
        "somecommand {input} {output}"
```

```
rule NAME:
    input:
        "path/to/inputfile",
        "path/to/other/inputfile"
    output:
        "path/to/outputfile",
        "path/to/another/outputfile"
    run:
        import shutil
        shutil.copyfile(input[0], output[0])

        with open(input[1], "r") as file:
            data = file.read()

        with open(output[1], "w") as file:
            file.write(data.upper())
```

Snakemake wildcards

- `{sample}` is a wildcard!
- here is how they work:

1. `input` wildcard: `{sample}`
    - The `input` section specifies the input files using the `"data/{sample}.txt"` pattern.
    - The `{sample}` wildcard matches any value and represents a variable part of the file name. For example, if you have input files named `data/A.txt` and `data/B.txt`, the `{sample}` wildcard will match `A` and `B` respectively.

2. `output` wildcard: `{sample}`
    - The `output` section specifies the output files using the `"results/{sample}_result.txt"` pattern.
    - Similar to the `input` wildcard, the `{sample}` wildcard in the output file pattern matches the same value as the input file wildcard. This ensures that the output file name is generated based on the value of `{sample}` from the corresponding input file.

```
rule example_rule:
    input:
        "data/{sample}.txt"
    output:
        "results/{sample}_result.txt"
    shell:
        "cat {input} | tr '[:lower:]' '[:upper:]' > {output}"
```

For example, if you have input files named `data/A.txt` and `data/B.txt`, Snakemake will generate the following input and output file combinations:

- 1.st execution:
    - Input: `"data/A.txt"`
    - Output: `"results/A_result.txt"`
- 2.nd execution:
    - Input: `"data/B.txt"`
    - Output: `"results/B_result.txt"`

snakemake: example I

- usually means, we have a set of input files, which we want to run through a program to get a set of output files
- e.g. our example I
- app.py does not have an input file, but it doesn't really matter

app.py

```python
import time
import random

time.sleep(3)

random_number = random.randint(0, 99)
print("result:", random_number)
```

snakemake: example I

Snakefile

```
results = "output_{i}.txt"

all_results = expand(results, i = [x for x in range(10)] )

rule all:
  input:
    all_results

rule run_app:
  input:
  output:
        "output_{i}.txt"
  shell:
        "python3 ../app.py  > {output}"
```

generic name of our output file
{i} is the wildcard

here we generate a list of all the output files

a rule, that requires some input, but doesn't do anything with it is a handy way to collect all final results we want

the rule, where all the work is done. no input needed, but it needs all the output_{i}.txt files

above, we have a dependency - "all" depends on "run_app", and can be used as a result collector!

how does it figure out how much the wildcard {i} is? from the all_results rule, which has the list of output_{i}.txt files as input.

basics of Snakemake

setting up snakemake

Workflow design: how-to

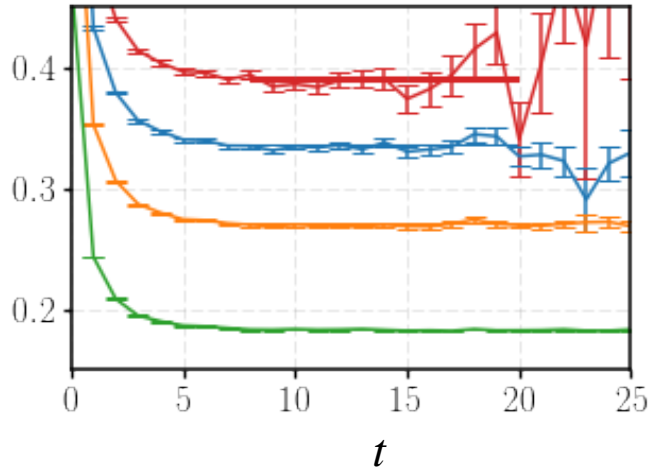
7 important steps when designing a workflow

- DEFINE THE OBJECTIVE:
  - identify the problem you are trying to solve
  - define the purpose and end goal of the workflow
- IDENTIFY THE TASKS:
  - break down the overall objective into smaller, manageable tasks
  - list the individual steps required to achieve the desired outcome
  - each task needs to be specific and well-defined
- DETERMINE THE TASK SEQUENCE:
  - determine the logical order of task execution
  - identify dependencies between tasks
- DESIGN TASK INTERFACES:
  - determine the flow of data (or outputs) between tasks
  - specify the inputs and outputs for each task
- WORKFLOW DIAGRAM:
  - visualize the workflow by creating a flowchart or diagram
- TEST AND ITERATE:
  - once the workflow is designed, run through it on a small scale
  - identify any issues or inefficiencies
  - refine the workflow based on above two points
- RUN:
  - keep the computer busy

Workflow Design: Example

- science based example (determining the mass and speed of composite particle)
  - data:
    - "correlation functions":
      $$C(\vec{p},t) = \sum_n c_n e^{-E_n t}$$
  - fitting data to determine
    $$E_n(\vec{p})$$
  - dispersion relation
    $$E_n(\vec{p}) = \sqrt{m^2 + \xi^2 \vec{p}\cdot\vec{p}}$$
    to determine m, ξ

$$\ln\left[\frac{C(\vec{p},t)}{C(\vec{p},t+1)}\right]$$



Workflow design: define the objective

questions for ourselves:

1. what do want to achieve?
2. what are we solving?
3. what do we want at the end?

1. we want to determine the mass and speed of a composite particle
2. we are extracting the mass and speed from a set of numerical data
3. two parameters, m and ξ, and the plot of $E_n(\vec{p}) = \sqrt{m^2 + \xi^2 \vec{p} \cdot \vec{p}}$

Workflow design: how-to

7 important steps when designing a workflow

- DEFINE THE OBJECTIVE:
  - identify the problem you are trying to solve
  - define the purpose and end goal of the workflow
- IDENTIFY THE TASKS:
  - break down the overall objective into smaller, manageable tasks
  - list the individual steps required to achieve the desired outcome
  - each task needs to be specific and well-defined
- DETERMINE THE TASK SEQUENCE:
  - determine the logical order of task execution
  - identify dependencies between tasks
- DESIGN TASK INTERFACES:
  - determine the flow of data (or outputs) between tasks
  - specify the inputs and outputs for each task
- WORKFLOW DIAGRAM:
  - visualize the workflow by creating a flowchart or diagram
- TEST AND ITERATE:
  - once the workflow is designed, run through it on a small scale
  - identify any issues or inefficiencies
  - refine the workflow based on above two points
- RUN:
  - keep the computer busy

Workflow design: identify the tasks

task 1: fit the correlators with varying models and ranges, determine the energies E_n at each $\vec{p}$

task 2: out of all the fits, pick the one with the smallest $\frac{\chi^2}{\mathrm{dof}}$

task 3: fit the set of $E_n(\vec{p})$ with $E_n(\vec{p}) = \sqrt{m^2 + \xi^2 \vec{p} \cdot \vec{p}}$

Workflow design: how-to

7 important steps when designing a workflow

- DEFINE THE OBJECTIVE:
  - identify the problem you are trying to solve
  - define the purpose and end goal of the workflow
- IDENTIFY THE TASKS:
  - break down the overall objective into smaller, manageable tasks
  - list the individual steps required to achieve the desired outcome
  - each task needs to be specific and well-defined
- DETERMINE THE TASK SEQUENCE:
  - determine the logical order of task execution
  - identify dependencies between tasks
- DESIGN TASK INTERFACES:
  - determine the flow of data (or outputs) between tasks
  - specify the inputs and outputs for each task
- WORKFLOW DIAGRAM:
  - visualize the workflow by creating a flowchart or diagram
- TEST AND ITERATE:
  - once the workflow is designed, run through it on a small scale
  - identify any issues or inefficiencies
  - refine the workflow based on above two points
- RUN:
  - keep the computer busy

Workflow design: task sequence & interfaces

- first task: **task 1**
  - input: raw data, fit model, fit range
  - output: E_n
    (with fit model type and fit range)
  - procedure: read raw data for each $\vec{p}$, construct χ^2 function, minimize χ^2, save $E_n(\vec{p})$
  - note: can run in parallel

Workflow design: task sequence & interfaces

- second task: **task 2**
  - input: list of E_n at various $\vec{p}$
  - output: E_n with smallest $\frac{\chi^2}{\mathrm{dof}}$
  - procedure: read all $E_n(\vec{p})$, chose one based on some criterion (smallest $\frac{\chi^2}{\mathrm{dof}}$)
  - depends on task 1

Workflow design: task sequence & interfaces

- third task: **task 3**
  - input: E_n for each $\vec{p}$
  - output: m, ξ, and a plot
  - procedure: read all $E_n(\vec{p})$, fit dispersion relation, determine m, ξ and construct plot
  - depends on task 2

Workflow design:
task sequence & interfaces

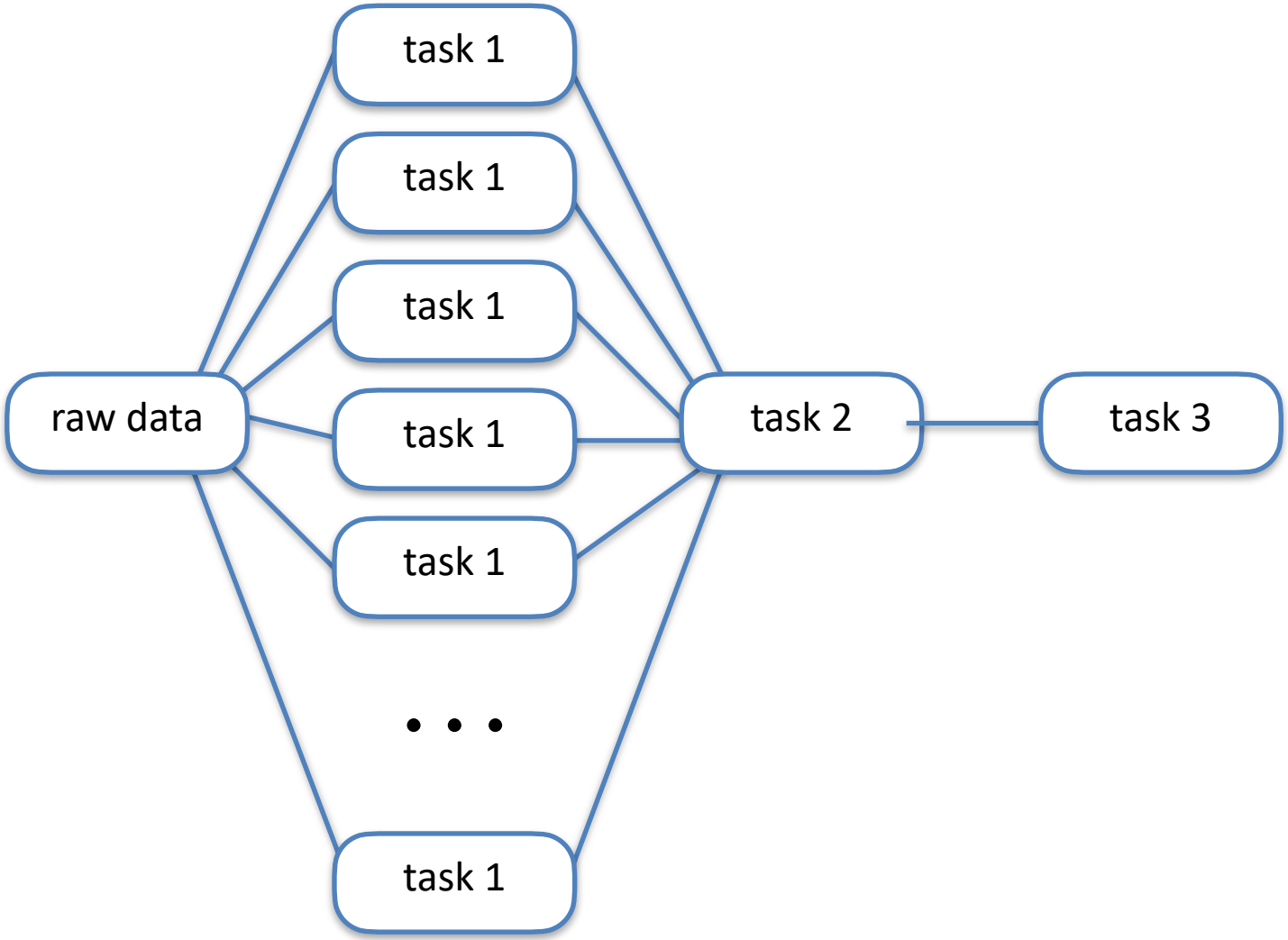
- first task: **task 1**
  - **program:** fit_correlator.py
- second task: **task 2**
  - **program:** combine_fits.py
- third task: **task 3**
  - **program:** fit_dispersion.py

Workflow design: how-to

7 important steps when designing a workflow

- DEFINE THE OBJECTIVE:
  - identify the problem you are trying to solve
  - define the purpose and end goal of the workflow
- IDENTIFY THE TASKS:
  - break down the overall objective into smaller, manageable tasks
  - list the individual steps required to achieve the desired outcome
  - each task needs to be specific and well-defined
- DETERMINE THE TASK SEQUENCE:
  - determine the logical order of task execution
  - identify dependencies between tasks
- DESIGN TASK INTERFACES:
  - determine the flow of data (or outputs) between tasks
  - specify the inputs and outputs for each task
- WORKFLOW DIAGRAM:
  - visualize the workflow by creating a flowchart or diagram
- TEST AND ITERATE:
  - once the workflow is designed, run through it on a small scale
  - identify any issues or inefficiencies
  - refine the workflow based on above two points
- RUN:
  - keep the computer busy

Workflow design: diagram



Workflow design: diagram

7 important steps when designing a workflow

- DEFINE THE OBJECTIVE:
  - identify the problem you are trying to solve
  - define the purpose and end goal of the workflow
- IDENTIFY THE TASKS:
  - break down the overall objective into smaller, manageable tasks
  - list the individual steps required to achieve the desired outcome
  - each task needs to be specific and well-defined
- DETERMINE THE TASK SEQUENCE:
  - determine the logical order of task execution
  - identify dependencies between tasks
- DESIGN TASK INTERFACES:
  - determine the flow of data (or outputs) between tasks
  - specify the inputs and outputs for each task
- WORKFLOW DIAGRAM:
  - visualize the workflow by creating a flowchart or diagram
- TEST AND ITERATE:
  - once the workflow is designed, run through it on a small scale
  - identify any issues or inefficiencies
  - refine the workflow based on above two points
- RUN:
  - keep the computer busy

the basic building blocks of Snakemake


```
rule RULE:
    input:
        "inis/input_1.ini"
    output:
        "tag/input_1.tag"
    shell:
        "./run_fit.sh {input} {output}"
```

- rule
  - 3 components
    - shell: here goes the command we want to execute
    - input: these is the input the shell command needs to run
    - output: this is the output the shell command produces
  - it's like in Makefile (if you are familiar)
  - the quantities from the input component are accessed in the shell command through {input}
  - the quantities from the output component are accessed in the shell command through {output}

the basic building blocks of Snakemake


```
import os
import glob
# list all files in inis subdir
ini_files = glob.glob("inis/*.ini")
print(ini_files)
```

- generic python code
  - can be anything: e.g. list generation

the basic building blocks of Snakemake


```
# snake file to run the pipeline
# run python3 ./fit_correlator.py --ini inis/input_x.ini && touch tag/input_x.tag
# for each input_x.ini in inis/

import os
import glob
# list all files in inis subdir
ini_files = glob.glob("inis/*.ini")
print(ini_files)

# rule generate tag files for each input file
rule all:
    input:
        expand("tag/input_{x}.tag", x=range(0, len(ini_files)))
    output:
        "correlators.done"
    shell:
        "touch {output}"

# rule run the fits
rule fit_correlator:
    input:
        ini="inis/input_{x}.ini"
    output:
        tag="tag/input_{x}.tag"
    shell:
        "./run_fit.sh {input.ini} {output.tag}"

# rule to cleanup
rule clean:
    shell:
        "rm -f tag/*.tag correlators.done"
```

- example Snakefile:
  - put python code anywhere - for legibility it's best if it's up top (FORTRAN logic)
  - several rules, but always try to have two:
    - rule all - as input you want all the outputs of the last last task you wish to automate
    - rule clean - to cleanup all the other stuff you have
  - rule fit_correlator - here is where all the hard work is done

the basic building blocks of Snakemake


```
# snake file to run the pipeline
# run python3 ./fit_correlator.py --ini inis/input_x.ini && touch tag/input_x.tag
# for each input_x.ini in inis/

import os
import glob
# list all files in inis subdir
ini_files = glob.glob("inis/*.ini")
print(ini_files)

# rule generate tag files for each input file
rule all:
    input:
        expand("tag/input_{x}.tag", x=range(0, len(ini_files)))
    output:
        "correlators.done"
    shell:
        "touch {output}"

# rule run the fits
rule fit_correlator:
    input:
        ini="inis/input_{x}.ini"
    output:
        tag="tag/input_{x}.tag"
    shell:
        "./run_fit.sh {input.ini} {output.tag}"

# rule to cleanup
rule clean:
    shell:
        "rm -f tag/*.tag correlators.done"
```

- example Snakefile:
  - first thing we do is generate a list of ini files located in the inis subdirectory (to keep the dir structure neat)
  - in rule all: we say that we want a file, called tag/input_{x}.tag generated for each file in the inis_file

    Snakemake needs to know how many things it runs. There are options how to do this:
    - hardcode
    - generate with python code (we use this one)
  - we used wildcards, so that each of these files in the inis directory can be run in parallel
  - the {x} between rule all and rule fit_correlator need not be named the same thing, but it's neat if we do so
  - run_fit.sh is a wrapper script, which we invoke so that we can add logging options into it without messing with the Snakefile (if we wish to do so)

the basic building blocks of Snakemake


```
# rule to run the correlator Snakefile
rule run_correlator:
    output:
        "correlators/correlators.done"
    shell:
        "cd correlators && snakemake -c 8"

# rule to run the combine Snakefile
rule run_combine:
    input:
        "correlators/correlators.done"
    output:
        "energies/energies.done"
    shell:
        "cd energies && snakemake -c 1"

# rule to run the dispersions Snakefile
rule run_dispersion:
    input:
        "energies/energies.done"
    output:
        "dispersion/dispersion_pion.pdf"
    shell:
        "cd dispersion && snakemake -c 1"

# Run all run rules from above
rule all:
    shell:
        "snakemake -j 8 run_correlator run_combine run_dispersion"

# clean rule
rule clean:
    shell:
        "cd correlators && snakemake -c 1 clean && cd ../energies && snakemake -c 1 clean && cd ../
        dispersion && snakemake -c 1 clean"
```

- can use Snakefile to invoke other Snakefile
- useful to split tasks into directories, set up the workflow for each separetely and then tie them together with a global Snakefile which invokes the others

- here run_correlator will use up to 8 cores (our hardcode choice)
- run_combine and run_dispersion will each use 1 core

- dependencies tell us that run_correlator can start without any input (no input specified), run_combine will only start when run_correlator is done and run_dispersion will only start when run_combine is done
- in rule all we just invoke snakemake (using the Makefile option of -j instead of -c)

The basics of Snakemake

paralelizing something simple

setting up Snakemake

- install mamba
  - wget https://github.com/conda-forge/miniforge/releases/latest/download/Mambaforge-Linux-x86_64.sh
  - chmod +x ./Mambaforge-Linux-x86_64.sh
  - ./Mambaforge-Linux-x86_64.sh
  - click yes to all the prompts, when asked whether to initialize by runnind conda init respond yes
  - relog
- create a new environment
  - # mamba create -n snakemake python=3.11
- activate
  - mamba activate snakemake
- install prerequisites:
  - mamba install numpy matplotlib iminuit h5py scipy
- install snakemake:
  - pip install snakemake

examples

- example I: git@github.com:leskovec/SnakeMake.intro.git
- example II: git@github.com:leskovec/SnakeMake.main.git

submitting a job - batch

submit.sh

```
#!/usr/bin/bash -l
#SBATCH -J snake
#SBATCH -o snake.o%j
#SBATCH -t 00:30:00
#SBATCH --ntasks=10
#SBATCH --nodes=1
#SBATCH --mem=500
#SBATCH --reservation=fri

source ~/.bashrc

mamba activate snakemake

snakemake -c 10 all
```

running in batch mode

- in this way, we request a node from SLURM and then make use of all the cores we have available

- pros:
  - no stress to SLURM
  - better utilization (now waiting in between)
- cons:
  - limited to number of cores on the node

submitting multiple jobs - cluster

- snakemake --slurm --default-resources slurm_account=leskovecl —jobs=10
- for our reservation: --cluster "sbatch --reservation=fri"
-

running in cluster mode

- in this way, we let Snakemake call slurm directly and submit jobs for us

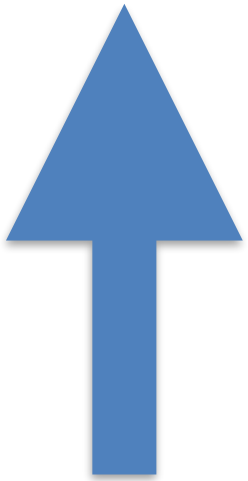
- pros:
  - bigger jobs
  - easier life
  - no limits
- cons:
  - stresses SLURM, can make admins mad :)
  - lazy way of setting up slurm array jobs :)

that’s the basics folks — play around and enjoy!

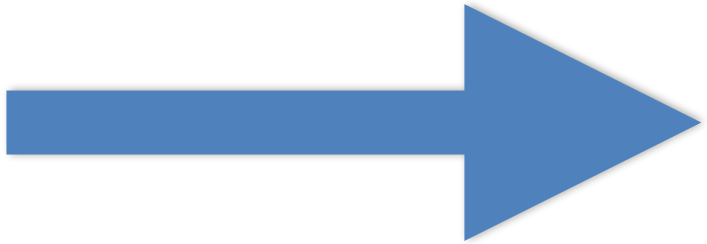
some additional online resources:

- https://docs.nersc.gov/jobs/workflow/snakemake/
- https://vincebuffalo.com/blog/2020/03/04/understanding-snakemake.html
- https://training.galaxyproject.org/training-material/topics/data-science/tutorials/snakemake/tutorial.html

Introduction to High-Throughput-Computing: SnakeMake



Introduction to High-Throughput-Computing: SnakeMake



Introduction to High-Throughput-Computing: SnakeMake