INTRODUCTION TO SNAKEMAKE

UE REPROHACKTHON

Thomas Cokelaer, Frédéric Lemoine
Institut Pasteur

2020/09/25

Institut Pasteur

PART 1

Introduction (Why Snakemake as a workflow manager)

1.1 What is a workflow/pipeline; what is a DAG ?

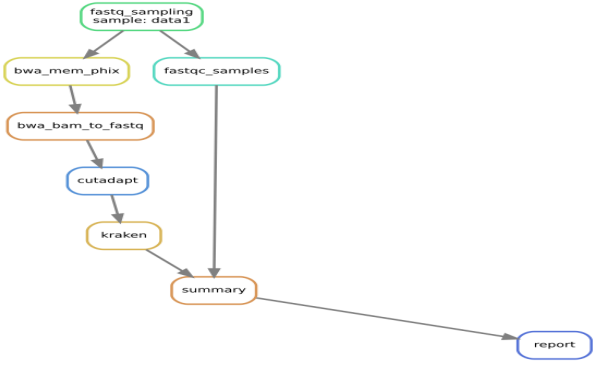


Figure: A workflow is a set of instructions that expect input/output

1.1 What is a workflow/pipeline; what is a DAG ?

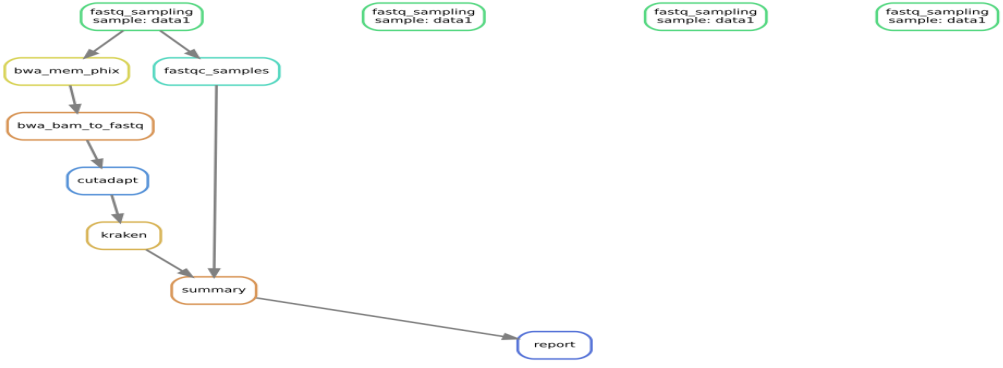


Figure: bioinformatics tools are often apply of several independent input files

1.1 What is a workflow/pipeline; what is a DAG ?

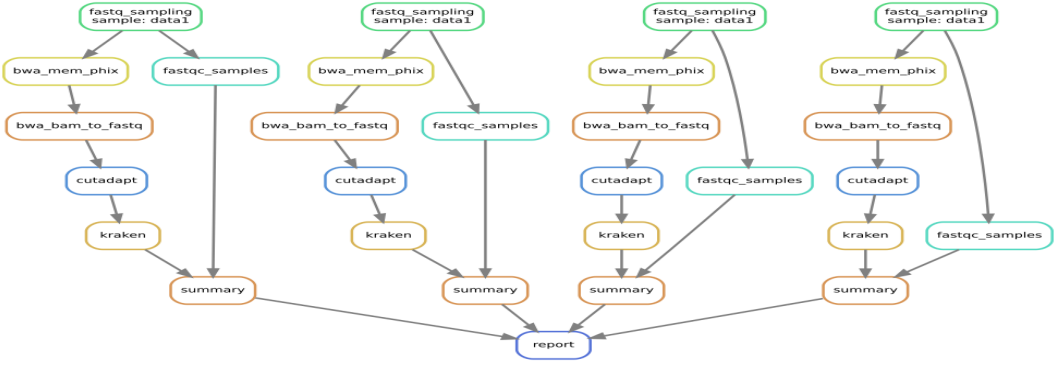


Figure: Ideal for embarassingly parallel problem

1.1 What is a workflow/pipeline; what is a DAG ?

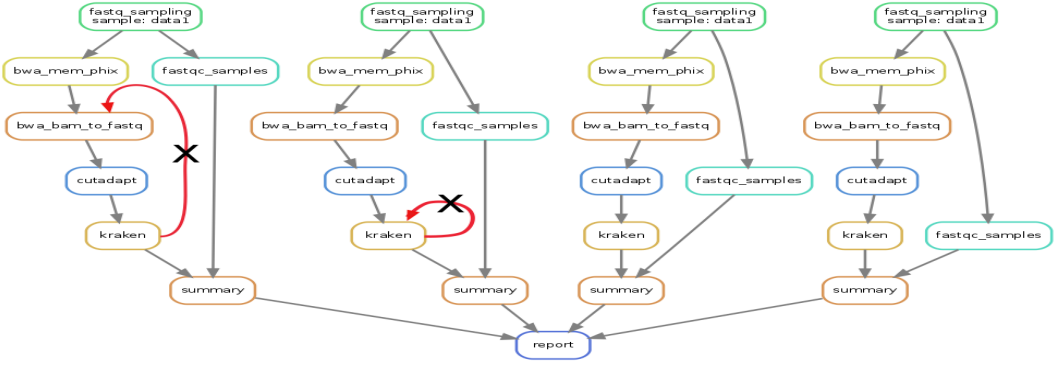


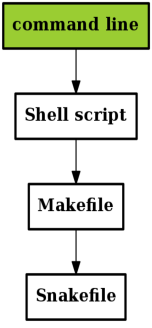
Figure: A DAG is a directed acyclic graph. Workflows usually requires a DAG (no self loop of feedback loop allowed !)

1.2 Toy example

Let us consider two FastQ files (independent samples) and let us map them on a reference (phiX174). The two sample files are named sample_A.fastq.gz and sample_B.fastq.gz

1.2 Toy example

The minimalist solution

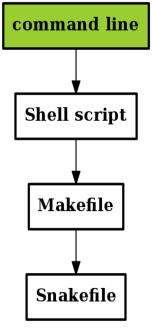


Shell commands: pretty simple

```
# Create a directory
mkdir -p mapped_sample

# Build the index of the reference
bwa index phiX174.fa

# Do the mapping twice on the two input FastQ files
bwa mem phiX174.fa A.fastq.gz | samtools view -Sb - >
    A.bam
bwa mem phiX174.fa B.fastq.gz | samtools view -Sb - >
    B.bam
```

1.2 Toy example

The minimalist solution

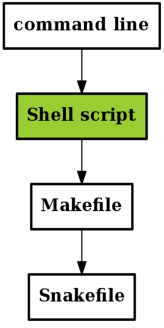


Shell commands: pretty simple

```
# Create a directory
mkdir -p mapped_sample

# Build the index of the reference
bwa index phiX174.fa

# Do the mapping twice on the two input FastQ files
bwa mem phiX174.fa A.fastq.gz | samtools view -Sb - >
    A.bam
bwa mem phiX174.fa B.fastq.gz | samtools view -Sb - >
    B.bam
```

Issues: Good start. Simple but what about some variables and scalability ?

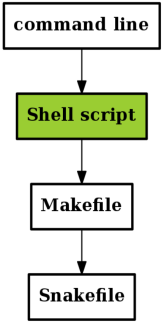
1.2 Toy example



Shell loop and variables

```
#!/bin/sh
REF="phiX174.fa"
ODIR="mapped_sample"
SAMPLES=`ls *.fastq.gz`

#Create a directory
mkdir -p $ODIR

# Build the index of the reference
bwa index $REF

# Do the mapping twice on the two input FastQ files
for var in $SAMPLES
do
  TARGET=${var/.fastq.gz/.bam}
  bwa mem $REF $SAMPLES | samtools view -Sb - > $ODIR/$TARGET
done
```

1.2 Toy example

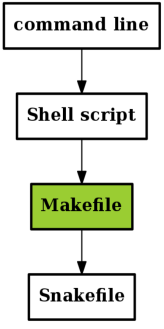


Shell loop and variables

```
#!/bin/sh
REF="phiX174.fa"
ODIR="mapped_sample"
SAMPLES='ls *.fastq.gz'

#Create a directory
mkdir -p $ODIR

# Build the index of the reference
bwa index $REF

# Do the mapping twice on the two input FastQ files
for var in $SAMPLES
do
  TARGET=${var/.fastq.gz/.bam}
  bwa mem $REF $SAMPLES | samtools view -Sb - > $ODIR/$TARGET
done
```

Issues: Still sequential. We start from scratch again and again.

1.2 Toy example



A Makefile consists of a set of rules in the following form

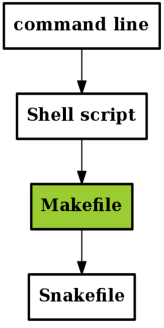
rule syntax

```
target: dependencies
    system command(s)
```

Makefile interests:
- handles the dependencies between rules
- avoids re-rerunning a task if the targets exist already

Widely used in C / C++ community for compilation of libraries.

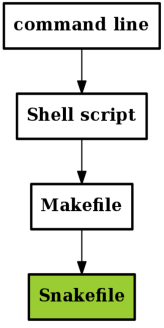
1.2 Toy example




```
SAMPLES = sample_A sample_B
ODIR = "mapped_sample"
FASTQS = $(patsubst %,%.fastq.gz,$(SAMPLES))
BAMS = $(patsubst %,$(ODIR)/%.bam,$(SAMPLES))
INDEX = phiX174.fa.bwt
REFERENCE = phiX174.fa

# Main rule
all: $(BAMS)

# bwa_mapping
$(ODIR)/%.bam: %.fastq.gz $(INDEX) $(ODIR)
  bwa mem $(REFERENCE) $< | samtools view -Sb - > $@

$(ODIR):
  mkdir -p $(ODIR)

# bwa_index
$(INDEX): $(REFERENCE)
  bwa index $<
```

1.2 Toy example

The Snakefile solution




```
SAMPLES = ["sample_A", "sample_B"]

rule all:
    input: expand("mapped_sample/{sample}.bam", sample=SAMPLES)

rule bwa_index:
    input: "phiX174.fa"
    output: "phiX174.fa.bwt"
    shell: "bwa index {input}"

rule bwa_mapping:
    input:
        ref = "phiX174.fa",
        index = "phiX174.fa.bwt",
        fastq = "{sample}.fastq.gz"
    output: "mapped_sample/{sample}.bam"
    shell:
        "bwa mem {input.ref} {input.fastq} | samtools view -Sb -
            > {output}"
```

1.2 Toy example

The Snakefile solution. Why is it better

Snakemake takes the best of Makefile:

- infers dependencies and execution order
- rules define obtain output files from input files
- structured pipelines

Snakemake combines its own syntax with Python syntax making it highly flexible

- Own domain specific syntax (rules and keywords)
- Use Python as the glue language
- The snakemake library itself is in Python

1.3 Snakemake interest

Python is a batteries included language.

1.3 Snakemake interest

Python is a batteries included language.

Snakemake as well !!

1.3 Snakemake interest

Python is a batteries included language.

Snakemake as well !!

- Clusters can be used with minimum efforts (no intrusive code)
- Workflows can be run from or up to a given rule
- Data provenance
- Nice logging system to follows the status
- Suspend / Resume
- Various code can be integrated: R, bash, and of course Python

PART 2

Snakemake Introduction (example 1)

2.1 workflow 1: counting lines in FastQ files

The goal is to count the number of reads in set of FastQ files. As a reminder here is the format of a FastQ files:

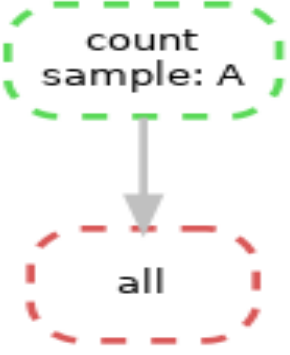
```
@HISEQ:426:C5T65ACXX:5:2301:5633:7203  1:N:0:GTGAAA
TTGCTTAATTCTATACTCATCCTCTACAGGGAGTTGGCAAGATTCAAAGACAACCAAAGAAGTCAAC
+
CCCFFFFFHHHHHJJJIIHIHIJJJJJJJJI@HEHIJJJJIIJJJJIAHIIJIIJJIJIIJJIIHGI
@HISEQ:426:C5T65ACXX:5:2301:5815:7120  1:N:0:GTGAAA
TATCTGCGATTGGTTCCATCTTCCCGGGTGACTGTGCAGCTGACTCCCATGCCCACCATCCCAGAGT
+
CCCFFFFFFHHHGHGGHIIJIIIJIJJIDFHIJIIHHIIIIJJIJJJIHCHIJIIJJJJJHHHHFF;
```

Each sequence read is described by 4 lines: the first line is the identifier starting with @, the second line is the DNA sequence, the third line (+ character) is a separator and the fourth line is the quality of each nucleotide.

2.2 concepts: rules, input/output

- **rule** is a keyword that defines a Snakemake step;
- The relation between rules is defined by their inputs/outputs
- Input/Output are files
- The first rule is called **all** by convention and is the last rule in your workflow. Therefore it has no output and input is the expected outcome of the pipeline

```
rule all:
    input:
        "stats/A.txt"

rule count:
    input: "A.fastq"
    output: "stats/A.txt"
    shell: "wc A.fastq > stats/A.txt"
```



Save the file into a file (usually called Snakefile)

2.3 Create the dag

To generate the image representing your workflow including all samples:

```
snakemake -s Snakefile --dag | dot -Tpng -o dag.png
```

If you have many samples, the dag is too complex, and a simpler representation is the rulegraph. It shows only the rules (excludig the true DAG with all samples)

```
snakemake -s Snakefile --rulegraph | dot -Tpng -o rg.
   png
```

2.4 Execution

To execute the pipeline,

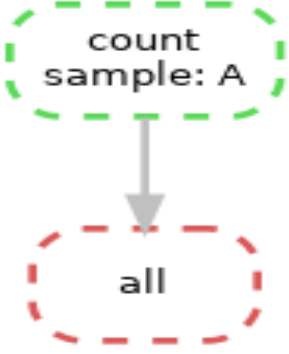
```
snakemake -s Snakefile --cores 1
```

This will create the file called A.txt in the directory ./stats Note that the directory is created automatically. If you run the pipeline again, nothing will happen. Indeed, the expected output file (stats/A.txt) exists already and the input file has not changed. If the input of a rule changes, the rule is executed again.

2.5 concepts: special keywords {input} / {output}

- In the shell command, we can reuse the variable names {input} and {output}. Note the accolades

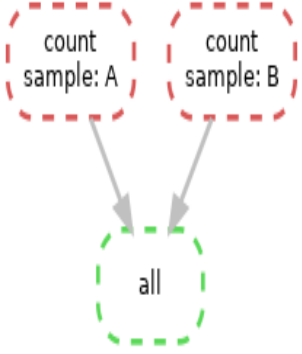
```
rule all:
    input:
        "stats/A.txt"

rule count:
    input: "A.fastq"
    output: "stats/A.txt"
    shell: "wc {input} > {output}"
```



2.6 concepts: the wildcards

- So far, we used only one sample and hardcoded its name. Not very useful. To parallelize, a nice mechanism is implemented in Snakemake: the wildcards.

```
samples = ['A', 'B']

rule all:
    input:
        "stats/A.txt", "stats/B.
            txt"

rule count:
    input: "{sample}.fastq"
    output: "stats/{sample}.txt"
    shell: "wc {input} > {output}"
```



2.7 concepts: the expand function

- With hundreds of file, we should use the expand function

```
samples = ['A', 'B']

rule all:
  input:
    expand("stats/{sample}.txt", sample=samples)

rule count:
  input: "{sample}.fastq"
  output: "stats/{sample}.txt"
  shell: "wc {input} > {output}"
```

2.8 final touch: count the reads, not the lines

Let us finalise the pipeline with a correct shell command

```
samples = ['A', 'B']

rule all:
  input:
    expand("stats/{sample}.txt", sample=samples)

rule count:
  input: "{sample}.fastq"
  output: "stats/{sample}.txt"
  shell:
    """
        wc {input} | awk ' END {{ print $1 / 4 }} ' -> {output}
    """
```

Note the duplication of the curly brackets. The actual syntax of awk uses only one curly brackets. In Snakemake, since the curly bracket is part of the syntax, it should be escaped (duplicated)

2.9 From shell to Python

Using Python in a rule is very simple: replace the **shell** with **run** keyword. No need for triple quotes (""") , just type valid Python code.

```
samples = ['A', 'B']

rule all:
  input:
    expand("stats/{sample}.txt", sample=samples)

rule count:
  input: "{sample}.fastq"
  output: "stats/{sample}.txt"
  run:
    count = 0
    with open(input[0], 'r') as fin:
        for line in fin.readlines():
            count += 1
    with open(output[0], "w") as fout:
        N = int(count / 4)
        fout.write("{}".format(N))
```

2.10 Execution

If output files are already present, you can force the execution of the entire pipelines:

```
snakemake -s Snakefile --cores 1 --forceall
```

Since we now have two samples, what about using 2 cores:

```
snakemake -s Snakefile --cores 2
```

2.11 From inline Python to external script

If your code is complex, you can use external script

```
samples = ['A', 'B']

rule all:
  input:
    expand("stats/{sample}.txt", sample=samples)

rule count:
  input: "{sample}.fastq"
  output: "stats/{sample}.txt"
  script: "script.py"
```

In the script you must use the snakemake namespace

```
count = 0
with open(snakemake.input[0], 'r') as fin:
    for line in fin.readlines(): count += 1
with open(snakemake.output[0], "w") as fout:
    N = int(count / 4) ; fout.write("{}".format(N))
```

2.12 onsuccess

Once all the rules are executed, you may have a final section. Usually, this is to create reports, indicates potential issues. It should not be long.

```
samples = ['A', 'B']

rule all:
  input:
    expand("stats/{sample}.txt", sample=samples)

rule count:
  input: "{sample}.fastq"
  output: "stats/{sample}.txt"
  script: "script.py"

onsuccess:
    # This is Python code here; you can use the shell() function
    import glob
    with open("out.html", "w") as fout:
        filenames = glob.glob("stats/*txt")
        for filename in filenames:
            res = open(filename, "r").read()
            fout.write("{}:{}<br>".format(filename, res))
```

PART 3

Snakemake in depth (example 2)

3.1 Example2: perform QC on FastQ files

So far we used a workflow with home-made script based (one thread). Here, we perform a QC of input fastq files with the standard tool **fastqc**. It takes a {}.fastq file as input and creates two files({}_fastqc.html and {}_fastqc.zip)

```
samples = ['A', 'B']

rule all:
    input:
        expand("fastqc/{sample}_fastqc.html", sample=samples)

rule fastqc:
    input: "{sample}.fastq.gz"
    output:
        html="fastqc/{sample}_fastqc.html",
        zip="fastqc/{sample}_fastqc.zip"
    shell: "fastqc {input} -o fastqc"
```

Note: here we have two outputs. It is a list of items separated by commas. You can give name (here html and zip)

3.2 log

When executing the previous workflow, fastqc prints lots of no essential information on the screen. Let us catch the stdout/stderr in a log file:

```
samples = ['A', 'B']

rule all:
    input:
        expand("fastqc/{sample}_fastqc.html", sample=samples)

rule fastqc:
    input: "{sample}.fastq.gz"
    output:
        html="fastqc/{sample}_fastqc.html",
        zip="fastqc/{sample}_fastqc.zip"
    log: "logs/{sample}.log"
    shell: "fastqc {input} -o fastqc >{log} 2>&1"
```

3.3 Threading

Disjoint paths in the DAG of jobs can be executed in parallel using --cores argument:

```
snakemake --cores 8
```

We can be more specific inside the rules:

```
rule fastqc:
    input: BLA
    output: BLI
    threads: 4
    shell: "fastqc -t {threads} {input} > {output}"
```

And use the same command:

```
snakemake --cores 8
```

but here only two jobs are executed at the same time

3.4 params

Usually applications or rules required some flexible parameters.

```
samples = ['A', 'B']
rule all:
    input:
        expand("fastqc/{sample}_fastqc.html", sample=samples)

rule fastqc:
    input: "{sample}.fastq.gz"
    output:
        html="fastqc/{sample}_fastqc.html",
        zip="fastqc/{sample}_fastqc.zip"
    params:
        nogroup=False
    threads: 4
    log: "logs/{sample}.log"
    run:
        if params.nogroup:
            shell("fastqc {input} -o fastqc -t {threads} --nogroup >{log}
                2>&1")
        else:
            shell("fastqc {input} -o fastqc -t {threads} >{log} 2>&1")
```

3.5 benchmarking

The benchmark directive takes a string that points to the file where benchmarking results shall be stored. It is possible to repeat a benchmark multiple times

```
samples = ['A', 'B']
rule all:
    input:
        expand("fastqc/{sample}_fastqc.html", sample=samples)

rule fastqc:
    input: "{sample}.fastq.gz"
    output:
        html="fastqc/{sample}_fastqc.html",
        zip="fastqc/{sample}_fastqc.zip"
    threads: 4
    benchmark:  repeat("benchmarks/{sample}.txt", 5)
    log: "logs/{sample}.log"
    shell: """fastqc {input} -o fastqc -t {threads} >{log} 2>&1 """
```

PART 4

Summary and resources

4.1 Evaluation

A job is executed if

- output file target does not exist
- output file needed by another executed job and does not exist
- input file newer than output file
- input file will be updated by other job
- execution is enforced

determined via breadth-first-search on DAG of jobs

4.2 Runs on clusters too

No intrusive code. It just worked on SGE and then on a SLURM cluster without changing a single line of code !

```
# execute the workflow on cluster with qsub
# submission command (up to 100 parallel jobs)
snakemake --cluster qsub --jobs 100

# execute the workflow with DRMAA
snakemake --drmaa --jobs 100

# execute the workflow on cluster with sbatch (
   SLURM)
snakemake --cluster "sbatch --qos fast" --jobs 100
```

4.3 Resources (memory)

We can be specific about memory used by a job with the resources keyword:

```
rule bwa_mapping:
    input: test.fastq
    output: test.bam
    threads: 4
    resources: mem_mb=1000
    shell: bwa mem -t {threads} {input} > {
       output}
```

and use the resources parameter when calling Snakemake:

```
# execute with only 8 cores and 1Gb memory

snakemake --cores 8 --resources mem_mb=1000
```

so here only one job at a time is executed

4.4 Errors

If an error occurs after hours of computation, fix the error in your code or missing files, and run snakemake again. Finished jobs won't be re-run.

4.5 Other features

- handles temporary and protected files
- run until a given rule
- run from a given rule
- stats about run time
- benchmark: run several times the rules
- any external scripts can be used (R, python, etc)
- remote files (http, ftp, google could, amazon, dropbox)
- rules may have priorities
- cluster time and memory can be fully customized
- modular: can include rules, or sub workflow

4.5 Other features

Conclusions

Mature

Snakemake is a mature tool ready for production.

batteries included

To cite just one great feature: free parallelization on a cluster.

Nice Syntax

The syntax is in Python, the library is in Python. Nevertheless, only a minimalist knowledge is required to get started since nice functions are already provided (e.g. expand).

Large community

Large snakemake community. See also the conda/bioconda community.

4.5 Other features

Discussions

Snakemake is great so what's wrong ?

Not much but here are some food for thoughts

- Snakefile uses Python syntax but Snakefile are not Python module
- Errors are sometimes too cryptic and definitely not useful for end-users
- Despite lots of sanity checks, if you are not careful you may end up in an infinite loop or delete the content of a file. So do lots of testing and save your data files before production. And just avoid symbolic links same input/output filenames.
- The rule syntax is great but developpers makes different choices on how they use them. So despite a great idea of sharing tools, you end up with many different pipelines and rules that does the same thing...

Institut Pasteur
25-28, rue du Docteur Roux
75724 Paris Cedex 15 - France
www.pasteur.fr