Snakemake overview

Thomas Cokelaer

Institut Pasteur

Nov 9th 2017
Snakemake and Sequana overview

Many bioinformatic pipeline frameworks available

A review of bioinformatic pipeline frameworks. Jeremy Leipzig Briefings in Bioinformatics, Volume 18, Issue 3, 1 May 2017, Pages 530–536, https://doi.org/10.1093/bib/bbw020

Many bioinformatic pipeline frameworks available

Figure: word cloud of 103 frameworks' description (less framework / pipeline / worfflow) https://github.com/pditommaso/awesome-pipeline

Which framework to choose ?

There are many frameworks out there.

Which framework to choose ?

There are many frameworks out there.

Some are **professional**, others not.

Which framework to choose ?

There are many frameworks out there.

Some are **professional**, others not.

Some are not **maintained** anymore or by a few developers.

Which framework to choose ?

There are many frameworks out there.

Some are **professional**, others not.

Some are not **maintained** anymore or by a few developers.

Many frameworks pass those filters. We have the luxury to choose one amongst many good frameworks !

So you need to define your requirements in terms of portability, language, reproducibility, parallelization, etc ?

Which framework to choose ?

- We need to be reactive
- Those days, developers code in R or Python.
- We need (non intrusive) parallelization in the NGS field

What about Snakemake ?

Python + GNU Makefile = Snakemake

Think Makefile, think DAG

Snakemake is a workflow manager

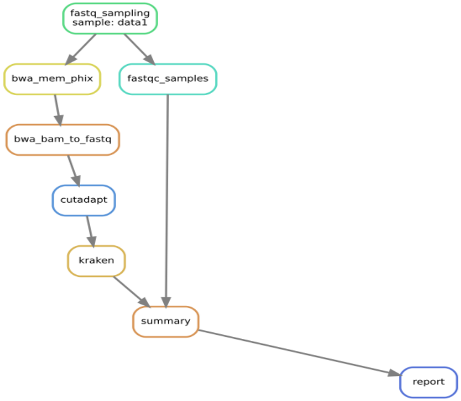


Figure: A *pipeline* problem

Think Makefile, think DAG

Snakemake is a workflow manager

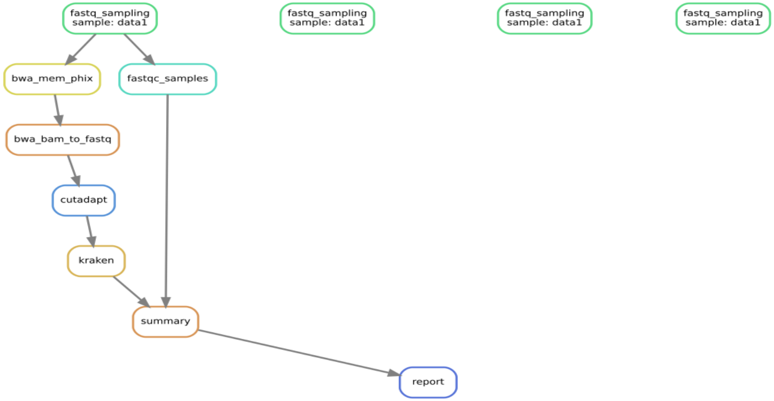


Figure: Ideal for embarassingly parallel problem

Think Makefile, think DAG

Snakemake is a workflow manager

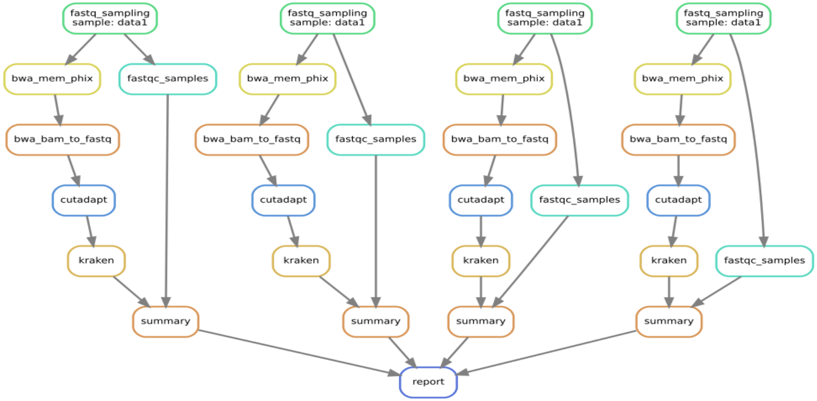


Figure: Ideal for embarassingly parallel problem

Think Makefile, think DAG

Snakemake is a workflow manager

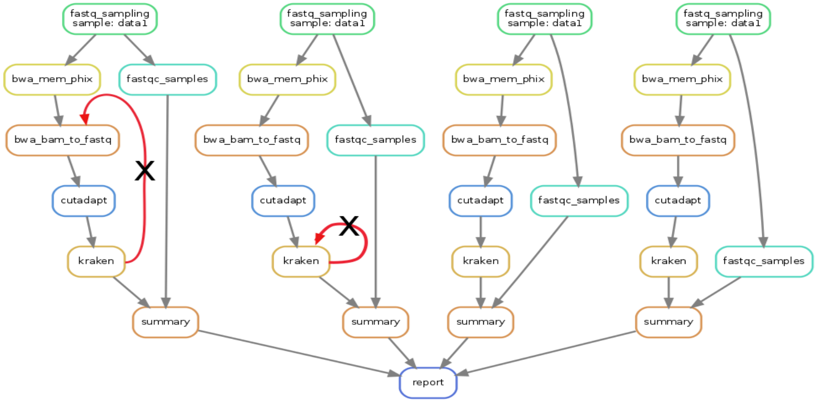


Figure: Requires a DAG (no self loop of feedback loop allowed !)

Why is it successful

Python is a batteries included language.

Why is it successful

Python is a batteries included language.

Snakemake as well !!

Why is it successful

Python is a batteries included language.

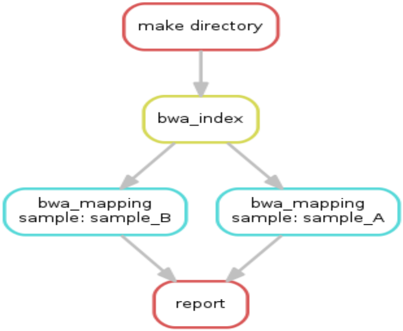
Snakemake as well !!

- 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

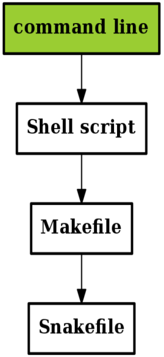
From sequential commands to dependent rule graph: a toy example

The problem

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



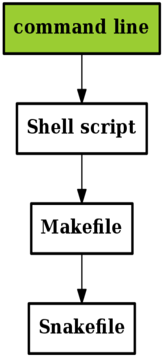
The minimalist solution



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
```

The minimalist solution



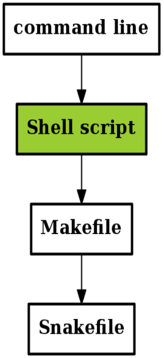
Shell commands: pretty simple

```
# 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 ?

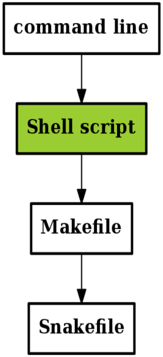
A shell solution



Shell loop and variables

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

#Create a directory
mkdir -p $ODIR

# Build the index of the reference
bwa index $REFERENCE

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

A shell solution



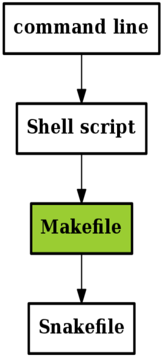
Shell loop and variables

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

#Create a directory
mkdir -p $ODIR

# Build the index of the reference
bwa index $REFERENCE

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

Issues

Still simple but sequential. What about dependencies between tasks ? What if a file exists already ? Do we start from scratch ?

The Makefile solution: a set of directives (rules)



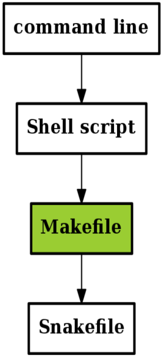
A Makefile consists of a set of rules in the following form

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.

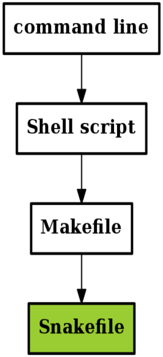
The Makefile solution: a set of directives (rules)



Makefile: a bwa_mapping and bwa_index rule

```
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 $<
```

The Snakefile solution



Snakefile

```
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 - > {o
```

The Snakefile solution

Snakemake logic: Makefile

Snakemake takes the best of Makefile:

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

Snakemake syntax: Python

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

Snakemake tutorial

The problem:

- convert a WAV signal into a frequency plot (spectrogram)
- Repeat for N input files
- Effect of the time window parameter W on the resolution
- Add an HTML / server report

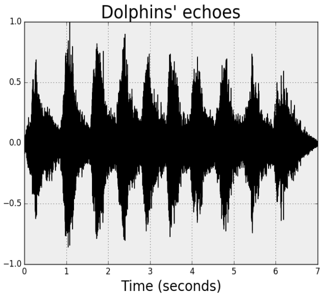


Figure: Input: a time series in WAV or OGG format

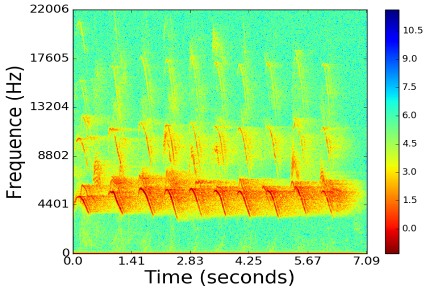


Figure: Output: a PNG image representing the time-frequency content

Rules

```
rule spectrogram:
    input:  "DOLPHINS.wav"
    output: "DOLPHINS.png"
    shell: "python spectrogram.py {input}"
```

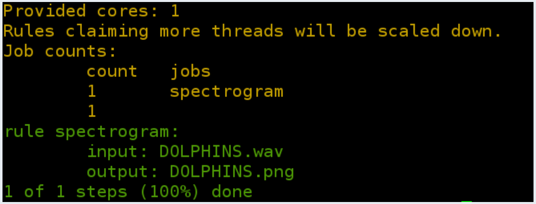
Save the file into a file called **Snakefile**

Execution

In a shell, type:

```
snakemake
```

Stdout



```
Provided cores: 1
Rules claiming more threads will be scaled down.
Job counts:
        count   jobs
        1       spectrogram
        1
rule spectrogram:
        input: DOLPHINS.wav
        output: DOLPHINS.png
1 of 1 steps (100%) done
```

Execution

From a shell:

```
snakemake -s gc_minimalist.rules
```

More options if a configuration file is required, or execution is on a cluster, or . . . something goes wrong.

Wildcards: generalized inputs/outputs

Wildcards can be used to generalize a rule.

Wildcards

```
rule spectrogram:
    input:  "{dataset}.wav"
    output: "{dataset}.png"
    shell: "python spectrogram.py {input}"
```

Wildcards: generalized inputs/outputs

Snakemake automatically resolved multiple named wildcards. You still need to set the final targets.

Wildcards !! does not work

```
rule all:
  input: ["DOLPHINS.png", "WHALES.png"]

rule spectrogram:
    input:  "{dataset}.wav"
    output: "{dataset}.png"
    shell: "python spectrogram.py {input}"
```

Wildcards: generalized inputs/outputs

If the rule's output matches a requested file, the substrings matched by the wildcards are propagated to the input files and to the variable wildcards

Wildcards + expands + variables

```
samples = ['DOLPHINS', 'WHALES']
rule all:
  input: expand("{dataset}.png",
                dataset=samples)

rule spectrogram:
    input:  "{dataset}.wav"
    output: "{dataset}.png"
    shell: "python spectrogram.py {input}"
```

Several wildcards

```
samples = ["DOLPHINS", "WHALES"]
windows = [512, 1024, 2048, 4096]

rule all:
    input: expand("{dataset}_{ws}.png",
                              dataset=samples,
                              ws=windows)

rule spectrogram:
    input: "{dataset}.wav"
    output: "{dataset}_{window}.png"
    shell:
      "python spec.py {input} {wildcards.window}
```

Config file

We can use a configuration file for parameters. Format are either JSON or YAML The community seems to prefer YAML.

config.yaml

```
samples: [DOLPHINS, WHALES]
windows: [512,1024,2048,4096]
```

Snakefile

```
configfile: "config.yaml"

rule all:
    input: expand("{dataset}_{ws}.png",
                            dataset=config['samples'],
                            ws=config['windows'])

rule spectrogram:
    input:  "{dataset}.wav"
    output: "{dataset}_{window}.png"
    shell:  "python spec.py {input} {wildcards.window}"
```

Add a rule without input/output

We could add a cleanup rule:

```
rule clean:
    shell: "rm -f DOL*png WHALE*png"
```

Since the rule does not produce any outputs and does not depend on other rules, it is not part of the workflow: the rule must be called explicitly:

```
snakemake clean
```

dependencies

If not target is specified, snakemake tries to apply the first rule in the workflow Do not put the clean rule at the beginning !

Handle logs

```
configfile: "config.yaml"

rule all:
  input: expand("data/{name}_{ws}.png",
                            name=config['samples'],
                            ws=config['windows'])

rule spectrogram:
  input: "data/{dataset}.wav"
  output: "data/{dataset}_{window}.png"
  log: "logs/{dataset}_{window}.log"
  shell:
    "python spec.py {input} {wildcards.window} > {log}"
```

The **param** and **run** keywords

```
rule all:
    input:
      expected_output_list,
      "server.ready"

rule server:
  output: touch("server.ready")
  params:
    port=config['port']
  run:  # Some python code
    from easydev import browse
    browse("http://127.0.0.1:%s" % params['port'])

rule spectrogram:
    ....
```

onsuccess section

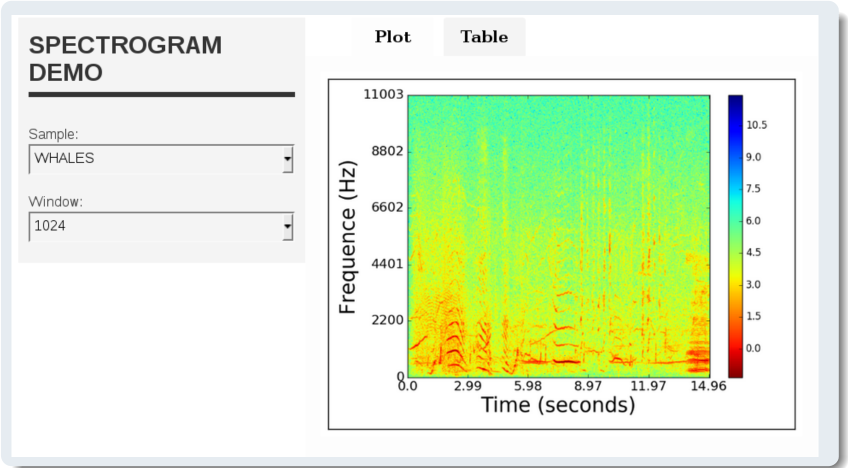
If the workflow succeeds, the onsuccess section is ran if provided (same for onerror section).

```
onsuccess:
    from myapp import SpecExample
    app = SpecExample(samples, window, "data")
    app.launch(port=config['port'])
```

You also have a **onerror** and **onstart** sections if needed.

demo

Get materials from
`https://github.com/sequana/presentations/tree/master/2016/snakemake_IPday_dec/examples/demo`



SPECTROGRAM DEMO

Sample: WHALES

Window: 1024

Advanced topics

alternative title: cheeries on the cake

Job execution (source: J. Koster talk)

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

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 bwa_mapping:
    input: test.fastq
    output: test.bam
    threads: 4
    shell: bwa mem -t {threads} {input} > {output}
```

And use the same command:

```
snakemake --cores 8
```

but here only two jobs are executed at the same time

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

Cluster execution

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
# (and up to 100 parallel jobs)
snakemake --cluster qsub --jobs 100

# tell the cluster system about the used threads
snakemake --cluster "qsub -pe threaded {threads}" --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
```

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.

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

Conclusions and discussions

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.

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...

References

- Great documentation for developers on Snakemake web site:
  - https://bitbucket.org/snakemake/snakemake/wiki/Documentation
  - https://bitbucket.org/snakemake/snakemake/wiki/Home
- Useful information:
  - http://slides.com/johanneskoester/deck-1
  - http://snakemake.bitbucket.org/snakemake-tutorial.html
  - http://slowkow.com/notes/snakemake-tutorial/
  - http://watson.nci.nih.gov/~sdavis/blog/flexible_bioinformatics_pipelines_with_snakemake/
- This talk and materials
  All snakefiles and data files to play around and available on
  https://github.com/sequana/presentations/2016/snakemake_IPday_dec

Snakemake Citations

Köster J., Rahmann S. Snakemake – a scalable bioinformatics workflow engine.

Bioinformatics application note Vol 28 (19) 2012

Questions

What happens when the snakemake is interrupted

- If you stop your snakemake (i.e. ctrl+c):

```
Terminating processes on user request.
Will exit after finishing currently running jobs.
Removing output files of failed job samtools since they might be corrupted:
reference.fa.fai
```

- On the cluster, the current job is not kill
- If you close your shell (Crash simulation):

```
IncompleteFilesException:
The files below seem to be incomplete. If you are sure that certain files are
not incomplete, mark them as complete with

    snakemake --cleanup-metadata <filenames>

To re-generate the files rerun your command with the --rerun-incomplete flag.
Incomplete files:
ERR036019_unsort.bam
```

- We can rerun the snakemake with `--rerun-incomplete`.