BIOINFORMATICS APPLICATION NOTE

Vol. 28 no. 19 2012, pages 2520–2522
doi:10.1093/bioinformatics/bts480

Genome analysis

Advance Access publication August 20, 2012

Snakemake—a scalable bioinformatics workflow engine

Johannes Köster[1,2,*] and Sven Rahmann[1]

[1]Genome Informatics, Institute of Human Genetics, University of Duisburg-Essen and [2]Paediatric Oncology, University Childrens Hospital, 45147 Essen, Germany

Associate Editor: Alfonso Valencia

ABSTRACT

Summary: Snakemake is a workflow engine that provides a readable Python-based workflow definition language and a powerful execution environment that scales from single-core workstations to compute clusters without modifying the workflow. It is the first system to support the use of automatically inferred multiple named wildcards (or variables) in input and output filenames.

Availability: http://snakemake.googlecode.com.

Contact: johannes.koester@uni-due.de

Received on May 14, 2012; revised on June 28, 2012; accepted on July 28, 2012

1 INTRODUCTION

Large-scale data analyses in bioinformatics involve the chained execution of many command line applications. Workflow engines help to automate these pipelines and ensure reproducibility.

Systems such as Biopipe (Hoon *et al.*, 2003), Taverna (Oinn *et al.*, 2004), Galaxy (Goecks *et al.*, 2010), GeneProf (Halbritter *et al.*, 2011) or PegaSys (Shah *et al.*, 2004) are easy to learn and use through their graphical user interface. Others such as Ruffus (Goodstadt, 2010), Pwrake (Tanaka and Tatebe, 2010), GXP Make (Taura *et al.*, 2010) and Bpipe (Sadedin *et al.*, 2012) use text-based definition of workflows, which can be advantageous: workflows can be edited without a graphical environment (e.g. directly on a remote server); and developers can collaborate on them through source code management tools. Similar to Pwrake and GXP Make, Snakemake is inspired by the build system GNU Make (Stallman and McGrath, 1991). They all infer the actual workflow (dependencies, parallelization) from a set of rules with input and output files. Snakemake complements these prior works with a syntax close to pseudocode, in the spirit of the Python language.

Snakemake interoperates with any installed tool or available web service with well-defined input and output (file) formats. Although this approach lacks type checking of intermediate files, it does not require tight integration of tools into the workflow system, such as with PegaSys (Shah *et al.*, 2004), and thus is most flexible. Snakemake itself is fully portable, as only a Python installation is required to run Snakefiles. It provides automatic scalability because it optimizes the number of parallel processes w.r.t. provided CPU cores and needed threads and can make use of single machines as well as cluster engines without modifying the workflow. In contrast to Pwrake and GXP Make, Snakemake does not rely on any password-less SSH setup or custom server processes running on the cluster nodes. Finally, Snakemake is the first system to support file name inference with multiple named wildcards in rules.

2 SNAKEMAKE LANGUAGE

A workflow is defined in a ‘Snakefile’ through a domain-specific language that is close to standard Python syntax. It consists of rules that denote how to create output files from input files. The workflow is implied by dependencies between the rules that arise from one rule needing an output file of another as an input file.

A rule definition specifies (i) a name, (ii) any number of input and output files and (iii) either a shell command or Python code that creates the output from the input. Input and output files may contain multiple named wildcards, whose values are inferred automatically from the files desired by the user. Listing 1 shows an example Snakefile for mapping sequence reads to a reference genome, which is a typical task in, e.g. cancer genomics (Meyerson *et al.*, 2010): paired-end sequence reads are given as .fastq files for four samples named 100–103 and mapped to the human reference genome. Then, a histogram of the per-nucleotide coverage is generated. There are two variable definitions (here SAMPLES and REF) that may also be included from external files or environment variables and five rules that each start with the keyword rule followed by a name and the definitions of input and output files and shell commands or Python code. Although Python code can be directly integrated into the workflow definition, Snakemake is not limited to Python scripts: any available tool or service may be invoked in a `shell-` or `run-`block and its output further processed.

The rule `fastq_to_sai` (l. 6–9) describes how to map the reads given the .fastq file and the reference. It uses two named wildcards, so it can be applied to the first and the second read of each read pair (wildcard `{group}`) from each sample (wildcard `{sample}`). If the rule is requested to create the file `100.1.sai`, the wildcard `{sample}` becomes 100 and `{group}` becomes 1, so that the input file `100.1.fastq` is expected. To resolve ambiguity, wildcards can be restricted to regular expressions (e.g. `{group,[12]}` only allows a single character from the set `{1,2}`, while `{group,\d+}` allows any number of numeric characters). Here, *BWA* (Li and Durbin, 2009) is used for read mapping, which produces suffix array interval (`.sai`) files that must be converted to the common format .bam for aligned reads; this is done by the rule `sai_to_bam` (l. 10–14). In the rule `fastq_to_sai`, input files are named (as `ref` and `reads`) and can be accessed as `{input.ref}` or `{input.reads}` in the shell block (l. 9).

*To whom correspondence should be addressed.

Listing 1. Example Snakefile for mapping paired-end reads with BWA.

```
 (1) SAMPLES = "100 101 102 103".split()
 (2) REF = "hg19.fa"
 (3) rule all:
 (4)   input: "{sample}.coverage.pdf".format(sample = sample)
 (5)     for sample in SAMPLES
 (6) rule fastq_to_sai:
 (7)   input: ref = REF, reads = "{sample}.{group}.fastq"
 (8)   output: temp("{sample}.{group}.sai")
 (9)   shell: "bwa aln {input.ref} {input.reads} > {output}"
(10) rule sai_to_bam:
(11)   input: REF, "{sample}.1.sai", "{sample}.2.sai",
(12)     "{sample}.1.fastq", "{sample}.2.fastq"
(13)   output: protected("{sample}.bam")
(14)   shell: "bwa sampe {input} | samtools view -Sbh - > {output}"
(15) rule remove_duplicates:
(16)   input: "{sample}.bam"
(17)   output: "{sample}.nodup.bam"
(18)   shell: "samtools rmdup {input} {output}"
(19) rule plot_coverage_histogram:
(20)   input: "{sample}.nodup.bam"
(21)   output: hist = "{sample}.coverage.pdf"
(22)   run:
(23)     from matplotlib.pyplot import hist, savefig
(24)     hist(list(map(int,
(25)       shell("samtools mpileup {input} | cut -f4",
(26)       iterable=True))))
(27)     savefig(output.hist)
```

In the rule sai_to_bam, all input files are accessed at once through {input} (l. 14). BWAs internal .sai files can be deleted automatically once the .bam files are created, which are in turn worth to be write-protected to avoid accidental deletion. Snakemake supports this by marking files as temp (l. 8) and protected (l. 13). The rule remove_duplicates removes polymerase chain reaction-induced duplicate reads from the .bam files using the 'samtools' package (Li *et al.*, 2009). Finally, the rule plot_coverage creates a coverage histogram for each position of the reference using Python's 'matplotlib' (Hunter, 2007). This rule illustrates how shell output from samtools is directly iterated over by Python code with Snakemake's built-in shell function in a run-block.

When Snakemake is invoked without a specific target, the first rule (here the input-only rule all) is executed. It ensures that the coverage plot and hence all needed intermediate files are created for each sample. See http://snakemake.googlecode.com for further examples and detailed documentation.

3 SNAKEMAKE ENGINE

Upon invocation, Snakemake creates a directed acyclic graph (DAG) that represents a plan of rule executions (Fig. 1). The nodes of the DAG are jobs (i.e. the execution of a rule), a directed edge between job A and B means that the rule underlying job B needs the output of job A as an input file. A path in the DAG represents a sequence of jobs that have to be executed serially. Importantly, two disjoint paths in the DAG can be executed independently from each other, i.e. in parallel. Since individual jobs can use multiple threads themselves, Snakemake can be instructed to solve a 0/1-knapsack problem to optimize the usage of CPUs, given a threshold of available cores. This mechanism allows to scale Snakemake to environments with a hard limit of used CPU cores, e.g. a shared compute server. Furthermore, using only as many threads as there are cores available can be beneficial for performance since it reduces the amount of context switching.

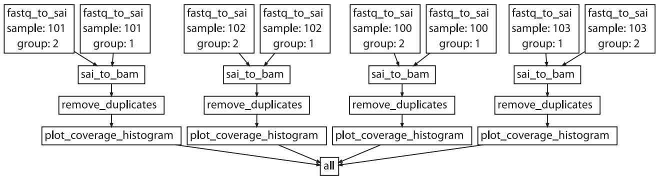
fastq_to_sai sample: 101 group: 2
fastq_to_sai sample: 101 group: 1
fastq_to_sai sample: 102 group: 2
fastq_to_sai sample: 102 group: 1
fastq_to_sai sample: 100 group: 2
fastq_to_sai sample: 100 group: 1
fastq_to_sai sample: 103 group: 1
fastq_to_sai sample: 103 group: 2
sai_to_bam
sai_to_bam
sai_to_bam
sai_to_bam
remove_duplicates
remove_duplicates
remove_duplicates
remove_duplicates
plot_coverage_histogram
plot_coverage_histogram
plot_coverage_histogram
plot_coverage_histogram
all

Fig. 1. DAG of jobs for the example workflow

By default, Snakemake only executes rules if the output files are not present or the modification time of the input files is newer. Together with the automatic deletion of output files from incomplete rule executions (e.g. due to a failing shell command), this enables Snakemake to avoid duplicate work when resuming workflows.

To analyse the workflow, Snakemake provides options to perform a dry-run without actual execution of jobs, give the reason for each executed job and print the DAG to the *graphviz*.dot format (Gansner and North, 2000) for visualization.

Apart from running on single machines, Snakemake contains a generic mechanism that allows the execution of jobs on a batch system or a compute cluster engine that is only constrained by the availability of a submit command that handles shell scripts (e.g. qsub) and a shared file system accessible by all cluster nodes. Hence, a Snakefile scales from single-core workstations over multi-core servers to compute clusters of different architectures, without the need to modify the workflow.

ACKNOWLEDGEMENTS

The authors thank Tobias Marschall (CWI Amsterdam) and Marcel Martin (TU Dortmund) for their tremendously helpful testing work, feature requests and comments.

Conflict of Interest: None declared.

REFERENCES

Gansner,E.R. and North,S.C. (2000) An open graph visualization system and its applications to software engineering. *Software Pract Exper*, **30**, 1203–1233.

Goecks,J. *et al.* (2010) Galaxy: a comprehensive approach for supporting accessible, reproducible, and transparent computational research in the life sciences. *Genome Biol.*, **11**, R86.

Goodstadt,L. (2010) Ruffus: a lightweight Python library for computational pipelines. *Bioinformatics*, **26**, 2778–2779.

Halbritter,F. *et al.* (2011) GeneProf: analysis of high-throughput sequencing experiments. *Nat. Methods*, **9**, 7–8.

Hoon,S. *et al.* (2003) Biopipe: a flexible framework for protocol-based bioinformatics analysis. *Genome Res.*, **13**, 1904–1915.

Hunter,J. (2007) Matplotlib: a 2D graphics environment. *Comput. Sci. Eng.*, **9**, 90–95.

Li,H. and Durbin,R. (2009) Fast and accurate short read alignment with Burrows–Wheeler transform. *Bioinformatics*, **25**, 1754–1760.

Li,H. *et al.* (2009) The sequence Alignment/Map format and SAMtools. *Bioinformatics*, **25**, 2078–2079.

Meyerson,M. *et al.* (2010) Advances in understanding cancer genomes through second-generation sequencing. *Nat. Rev. Genet.*, **11**, 685–696.

Oinn,T. *et al.* (2004) Taverna: a tool for the composition and enactment of bioinformatics workflows. *Bioinformatics*, **20**, 3045–3054.

Sadedin,S.P. *et al.* (2012) Bpipe: a tool for running and managing bioinformatics pipelines. *Bioinformatics*, **28**, 1525–1526.

Shah,S.P. *et al.* (2004) Pegasys: software for executing and integrating analyses of biological sequences. *BMC Bioinformatics*, **5**, 40.

Stallman,R.M. and McGrath,R. (1991) *GNU Make—A Program for Directing Recompilation*. http://wwwgnu.org/software/make/.

Tanaka,M. and Tatebe,O. (2010) Pwrake: a parallel and distributed flexible workflow management tool for wide-area data intensive computing. In *HPDC'10,* ACM. pp. 356–359. http://dl.acm.org/citation.cfm?id = 1851529.

Taura,K. *et al.* (2010) Design and Implementation of {GXP} Make – A Workflow System Based on Make. In *IEEE International Conference on eScience*, IEEE Computer Society, pp. 214–221, Los Alamitos, CA, USA.