



INSTITUTE  
FOR RESEARCH  
IN BIOMEDICINE

# The PanPipe Software Package

---

Daniel Ortiz

February 13, 2019

# Table of Contents

---

1. Introduction
2. Package Overview
3. Main Tools and File Formats
4. Toy Pipeline Example
5. Current Status and Future Work

# Introduction

---

- Pipeline execution is a complex task
  - Pipeline composed of very heterogeneous tasks/steps
  - Steps may present dependencies with other ones
  - Often necessary to add or remove pipeline steps
  - Need to allocate computational resources
  - Independent steps should be executed concurrently
  - Hard to maintain and reuse code
  - ...
- PanPipe has been created as a highly portable, configurable and extensible solution

# Package Overview

---

# Package Dependencies

- Shell Bash
- Python
- Slurm Workload Manager (optional)

# Package Installation

- Obtain the package using git:

```
git clone gitlab@fsupeksvr.irbbarcelona.pcb.ub.es:dortiz/panpipe.git
```

- Change to the directory with the package's source code and type:

```
./reconf  
./configure  
make  
make install
```

**NOTE:** use `--prefix` option of `configure` to install the package in a custom directory

- PanPipe is an engine to execute general pipelines
- Executes only those pipeline steps that are pending
- Handles computational resources for each step
- Executes job arrays



# Execution Model

- PanPipe is based on the *flow-based programming* paradigm
  - Network of *black box* processes
  - Relations between processes are defined by the data they exchange
  - Component oriented
- PanPipe follows a simple execution model based on a file enumerating a list of pipeline steps to be executed
- Steps are executed simultaneously unless dependencies are specified
- Step implementation is given in module files

# Main Tools and File Formats

---

- `pipe_exec`
- `pipe_exec_batch`
- `pipe_check`
- `pipe_status`

- Automates execution of general pipelines
- Main input parameters:
  - `--pfile <string>`: file with pipeline steps to be performed
  - `--outdir <string>`: output directory
  - `--sched <string>`: scheduler used for pipeline execution
  - `--showopts`: show pipeline options
  - `--checkopts`: check pipeline options
  - `--debug`: do everything except launching pipeline steps

- Content of output directory:
  - `scripts`: directory containing the scripts used for each pipeline step
  - `<pipeline_step_name>`: directory containing the results of the pipeline step of the same name
- Additional directories may be created depending on the pipeline

- **Built-in Scheduler**

- Allows to execute pipelines locally
- Incorporates a basic resource allocation mechanism

- **Slurm Scheduler**

- Allows to exploit large computational resources
- Usage transparent to the user
- Slurm behavior influenced by pipeline description

- Automates execution of pipeline batches
- Main input parameters:
  - -f <string>: file with a set of pipe\_exec commands
  - -m <string>: Maximum number of concurrently executed pipelines
  - -o <string>: Output directory to move output of each pipeline

- Checks correctness of pipelines and converts them to other formats
- Main input parameters:
  - -p <string>: pipeline file
  - -g: print pipeline in graphviz format



- Checks execution status of a given pipeline
- Main input parameters:
  - -d <string>: directory where the pipeline steps are stored
  - -s <string>: step name whose status should be determined (optional)

# The `panpipe_lib.sh` Library

- Shell library with functions used by the previously described tools
- Functions can be classified as follows:
  - Implementation of the package execution model
  - Automated creation of scripts executing pipeline steps
  - Helper functions to implement pipeline steps

- **Pipeline file:** file enumerating all of the pipeline steps to be carried out when processing a normal-tumor sample
- **Module file:** file defining the code of the pipeline steps
- **Pipeline automation script:** file with a sequence of `pipe_exec` commands automating the analysis of a dataset

- **Module import** (module names separated by commas)
- **Entry format** (one entry per line)

Step name, Slurm account, Slurm partition, CPUs, Memory limit, Time limit, Dependencies, ...

- **Dependency types:** none, after, afterok, afternotok, afterany

```
#import pipe_software_test
#
step_a cpus=1 mem=32 time=00:01:00 stepdeps=none
step_b cpus=1 mem=32 time=00:01:00 stepdeps=afterok:step_a
step_c cpus=1 mem=32 time=00:01:00 throttle=2 stepdeps=afterok:step_a
```

- Contains the definition of the different steps
- Written in `bash`
- Three `bash` functions should be defined for each step:
  - `stepname_explain_cmdline_opts()`
  - `stepname_define_opts()`
  - `stepname()`

## Module File: `stepname_explain_cmdline_opts()`

- This function documents the command line options that the step needs to work
- The aggregated documentation for the different steps is shown when executing `pipe_exec --showopts`
- Whenever two steps share the same option, it is important to give it the same name

## Module File: stepname\_explain\_cmdline\_opts()

```
step_a_explain_cmdline_opts()
{
    # -a option
    description="Sleep time in seconds for step_a (required)"
    explain_cmdline_opt "-a" "<int>" "$description"
}
```

## Module File: `stepname_define_opts()`

- This function should create a string containing the options that are specific to the step
- The main idea is to map command line options to step options
- The package provides multiple built-in functions to make the implementation of this function easier



## Module File: stepname\_define\_opts()

```
stepname_define_opts()
{
    # Initialize variables
    local cmdline=$1
    local jobspec=$2
    local optlist=""

    # Use built-in functions to add options to optlist variable
    ...

    # Save option list
    save_opt_list optlist
}
```

## Module File: stepname\_define\_opts()

```
step_a_define_opts()
{
    # Initialize variables
    local cmdline=$1
    local jobspec=$2
    local optlist=""

    # -a option
    define_cmdline_opt "$cmdline" "-a" optlist || exit 1

    # Save option list
    save_opt_list optlist
}
```

## Module File: `stepname()`

- Implements the step
- The function should incorporate code at the beginning to read the options defined by `stepname_define_opts()`

## Module File: stepname()

```
step_a()
{
    # Initialize variables
    local sleep_time=`read_opt_value_from_line "$*" "-a"`

    # Sleep some time
    sleep ${sleep_time}
}
```

# Pipeline Automation Script

- Automates the analysis of a whole dataset
- At each entry (one per line), `pipe_exec` tool is used to execute a whole pipeline
- Can be used as input for `pipe_exec_batch`
- Entry example:

```
pipe_exec --pfile example.ppl --outdir outdir1 --sched SLURM -opt1 <opt1_val> -opt2 <opt2_val> ...  
pipe_exec --pfile example.ppl --outdir outdir2 --sched SLURM -opt1 <opt1_val> -opt2 <opt2_val> ...  
pipe_exec --pfile example.ppl --outdir outdir3 --sched SLURM -opt1 <opt1_val> -opt2 <opt2_val> ...  
...  
pipe_exec --pfile example.ppl --outdir outdirn --sched SLURM -opt1 <opt1_val> -opt2 <opt2_val> ...
```

- Since multiple imports are permitted, a new module may contain step definitions missing in another one
- The order in which modules are imported is relevant
  - if two modules define the same function, the definition in the module imported last will prevail
  - the previous property can be used to modify a specific step without repeating the code of the whole module

# Toy Pipeline Example

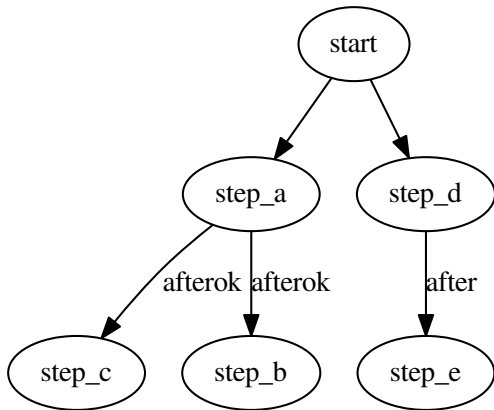
---

# Pipeline File

```
#import pipe_software_test
#
step_a cpus=1 mem=32 time=00:01:00 stepdeps=None
step_b cpus=1 mem=32 time=00:01:00 stepdeps=afterok:step_a
step_c cpus=1 mem=32 time=00:01:00 throttle=2 stepdeps=afterok:step_a
step_d cpus=1 mem=32 time=00:01:00 stepdeps=None
step_e cpus=1 mem=32 time=00:01:00 stepdeps=after:step_d
```



# Pipeline



## Current Status and Future Work

---

# Current Status

	PanPipe	Nextflow	Galaxy	Toil	Snakemake	Bpipe
Platform	Bash	Groovy/JVM	Python	Python	Python	Groovy/JVM
Native task support	Yes(bash)	Yes(any)	No	No	Yes(bash)	Yes(bash)
Common workflow lang.	No	No	Yes	Yes	No	No
Streaming processing	Yes	Yes	No	No	No	No
Dynamic branch eval.	Yes	Yes	?	Yes	Yes	Undocumented
Code sharing integration	No	Yes	No	No	No	No
Workflow modules	Yes	No	Yes	Yes	Yes	Yes
Workflow versioning	Yes	Yes	Yes	No	No	No
Automatic error failover	Yes	Yes	No	Yes	No	No
Graphical user interface	No	No	Yes	No	No	No
DAG rendering	Yes	Yes	Yes	Yes	Yes	Yes

Based on (Di Tommaso et al. 2017)

- Increase reproducibility of results
  - Support for conda
  - Support for containers
- Extend documentation
- Improve design and usability

## References



Di Tommaso, Paolo, Maria Chatzou, Evan W. Floden, Pablo P. Barja, Emilio Palumbo, and Cedric Notredame (2017). “Nextflow enables reproducible computational workflows”. In: *Nature Biotechnology* 35.4, pp. 316–319. ISSN: 1087-0156. DOI: 10.1038/nbt.3820. URL: <http://dx.doi.org/10.1038/nbt.3820>.