slurmpter

Release 1.0.0

Isaac Chun Fung WONG

Oct 30, 2020

CONTENTS:

Slurmpter (Slurm Scripter) is a package to build Slurm submit files of a workflow of jobs easily. The package uses PyCondor as the backend. The user interface of slurmpter is very similar to that of PyCondor except for some arguments dedicated for Slurm.

CHAPTER

ONE

INSTALLATION

1.1 PyPI

The latest release of `slurmpter` can be installed with `pip`:

```
pip install slurmpter
```

1.2 Conda

The latest release of `slurmpter` can be installed with `conda`:

```
conda install -c conda-forge slurmpter
```

1.3 Development Version

The latest development version of `slurmpter` can be installed from GitLab:

```
pip install git+https://gitlab.com/isaac-cfwong/slurmpter.git
```

1.4 Dependencies

- Python >= 3.5
- pycondor

CHAPTER

TWO

TUTORIAL

2.1 Build a simple workflow with only one job

Suppose now you want to submit a single job to Slurm, the executable is `<exec>` with arguments `<args>`.

```
from slurmpter import SlurmJob

# Define the job directories to write submit files, standard outputs and standard␣
↪errors.
submit = "slurm/submit"
output = "slurm/output"
error = "slurm/error"

# Construct a SlurmJob object to define the job.
job = SlurmJob(name="job", executable="<exec>", submit=submit, output=output,␣
↪error=error)
job.add_arg("<args>")

# Call build() to build the submit files but do not submit the jobs immediately.
job.build()
# Call build_submit() to build the submit files and submit the jobs sequentially.
#job.build_submit()
```

If you want to run a Python script e.g. `script.py`, replace `<exec>` with `python` and `<args>` with `script.py`.

2.2 Build a simple workflow with mutiple jobs

Suppose now you want to submit two jobs with different executables in a batch to Slurm,

```
from slurmpter import Slurm, SlurmJob

# Define the job directories to write submit files, standard outputs and standard␣
↪errors.
submit = "slurm/submit"
output = "slurm/output"
error = "slurm/error"

# Construct a Slurm object to hold all the jobs.
slurm = Slurm(name="slurm", submit=submit)
# Construct a SlurmJob object to define the first job.
job_0 = SlurmJob(name="job_0", executable="<exec_0>", submit=submit, output=output,␣
↪error=error, slurm=slurm)
```

(continues on next page)

(continued from previous page)

```
job_0.add_arg("<args_0>")

# Construct another SlurmJob object to define the second job.
job_1 = SlurmJob(name="job_1", executable="<exec_1>", submit=submit, output=output,␣
↪error=error, slurm=slurm)
job_1.add_arg("<args_1>")

# Call build() to build the submit files but do not submit the jobs immediately.
slurm.build()
# Call build_submit() to build the submit files and submit the jobs sequentially.
#slurm.build_submit()
```

Note that `job_0` and `job_1` do not have any inter-dependency, they can run concurrently if resource is available.

2.3 Complex workflow with job inter-dependencies

Please refer to *example* for a more complex workflow.

CHAPTER

THREE

API

3.1 SlurmJob

class slurmpter.**SlurmJob**(*name*, *executable*, *error=None*, *output=None*, *submit=None*, *nodes=None*, *ntasks_per_node=None*, *cpus_per_task=None*, *mem_per_node=None*, *extra_sbatch_options=None*, *extra_srun_options=['ntasks=1', 'exclusive']*, *extra_lines=None*, *modules=None*, *slurm=None*, *arguments=None*, *verbose=0*)

A class for handling Slurm job.

Methods:

__init__(name, executable[, error, output, ...])	Constructor.
add_arg(arg[, name, retry])	Add argument to Job
add_args(args)	Adds multiple arguments to Job
add_child(node)	Adds node to children list
add_children(nodes)	Adds nodes to the children list
add_parent(node)	Adds node to parents list
add_parents(nodes)	Adds nodes to the parents list
build([makedirs, fancyname])	Build and save the submit file for Job.
build_submit([makedirs, fancyname, ...])	Build and submit sequentially.
haschildren()	Checks for any children nodes
hasparents()	Checks for any parent nodes
submit_job([submit_options])	Submit Job to Slurm.

__init__(*name*, *executable*, *error=None*, *output=None*, *submit=None*, *nodes=None*, *ntasks_per_node=None*, *cpus_per_task=None*, *mem_per_node=None*, *extra_sbatch_options=None*, *extra_srun_options=['ntasks=1', 'exclusive']*, *extra_lines=None*, *modules=None*, *slurm=None*, *arguments=None*, *verbose=0*)

Constructor.

Parameters

- **name** (*str*) – Name of the job.
- **executable** (*str*) – Path of the executable for the job.
- **error** (*str, optional*) – Directory of error files.
- **output** (*str, optional*) – Directory of output files.
- **submit** (*str, optional*) – Directory of submit files.
- **nodes** (*str, optional*) – <minnodes[-maxnodes]> Request that a minimum of

minnodes nodes be allocated to this job. A maximum node count may also be specified with maxnodes. If only one number is specified, this is used as both the minimum and maximum node count.

- **ntasks_per_node** (*str, optional*) – <ntasks> Request that ntasks be invoked on each node.
- **cpus_per_task** (*str, optional*) – <ncpus> Request that ncpus be allocated per process.
- **mem_per_node** (*str, optional*) – <size[units]> Specify the real memory required per node.
- **extra_sbatch_options** (*array-like str, optionl*) – An array of extra options to append after '#SBATCH '.
- **extra_srun_option** (*array-like str, optional*) – An array of extra options to append after 'srun'.
- **extra_lines** (*array-like str, optional*) – An array of extra lines to add before srun.
- **modules** (*array-like str, optional*) – An array of modules to append after 'module load '.
- **slurm** (Slurm, *optional*) – If specified, SlurmJob will be added to Slurm.
- **arguments** (*str or iterable, optional*) – Arguments to initialize the job list.
- **verbose** (*int, optional*) – Level of logging verbosity option are 0-warning, 1-info, 2-debugging (default is 0).

add_arg (*arg*, *name=None*, *retry=None*)
Add argument to Job

Parameters

- **arg** (*str*) – Argument to append to Job args list.
- **name** (*str or None, optional*) – Option to specify a name related to this argument. If a name is specified, then a separate set of log, output, and error files will be generated for this particular argument (default is None).

  New in version 0.1.2.
- **retry** (*int or None, optional*) – Option to specify the number of times to retry this node. Default number of retries is 0. Note: this feature is only available to Jobs that are submitted via a Dagman.

  New in version 0.1.2.

Returns **self** – Returns self.

Return type object

add_args (*args*)
Adds multiple arguments to Job

Parameters **args** (*iterable*) – Iterable of arguments to append to the arguments list

Returns **self** – Returns self.

Return type object

add_child (*node*)
Adds node to children list

Parameters **node** (*BaseNode*) – Job or Dagman to append to the children list.

Returns **self** – Returns self.

Return type object

add_children(*nodes*)
Adds nodes to the children list

Parameters **nodes** (*list or tuple*) – List of nodes to append to the children list

Returns **self** – Returns self.

Return type object

add_parent(*node*)
Adds node to parents list

Parameters **node** (*BaseNode*) – Job or Dagman to append to the parents list.

Returns **self** – Returns self.

Return type object

add_parents(*nodes*)
Adds nodes to the parents list

Parameters **nodes** (*list or tuple*) – List of nodes to append to the parents list

Returns **self** – Returns self.

Return type object

build(*makedirs=True*, *fancyname=True*)
Build and save the submit file for Job.

Parameters

- **makedirs** (*bool, optional*) – Create job directories if not exist.
- **fancyname** (*bool, optional*) – Append the name with date and unique id.

Returns **self** – Self object.

Return type object

build_submit(*makedirs=True*, *fancyname=True*, *submit_options=None*)
Build and submit sequentially.

Parameters

- **makedirs** (*bool, optional*) – Create directories if not exist.
- **fancyname** (*bool, optional*) – Append date of unique id to submit.
- **submit_options** (*str, optional*) – Options to be passed to 'sbatch'.

Returns **self** – Self object.

Return type object

haschildren()
Checks for any children nodes

Returns Returns whether or not this node has any child nodes.

Return type bool

hasparents()
Checks for any parent nodes

Returns Returns whether or not this node has any parent nodes.

Return type bool

submit_job(*submit_options=None*)
Submit Job to Slurm.

Parameters **submit_options** (*str, optional*) – Submit options appends after sbatch.

Returns **self** – Self object.

Return type object

3.2 Slurm

class slurmpter.**Slurm**(*name*, *submit=None*, *extra_lines=None*, *verbose=0*)
Slurm object manages the workflow of a series of SlurmJobs.

Methods:

__init__(name[, submit, extra_lines, verbose])	Constructor.
add_job(job)	Add job to Slurm.
build([makedirs, fancyname])	Build slurm submit files.
build_submit([makedirs, fancyname, ...])	Build and submit sequentially.
submit_slurm([submit_options])	Submit to slurm.
visualize([filename])	Visualize Slurm graph.

__init__(*name*, *submit=None*, *extra_lines=None*, *verbose=0*)
Constructor.

Parameters

- **name** (*str*) – Name of the Dagman instance.
- **submit** (*str*) – Directory to write submit files.
- **extra_lines** (*array-like str*) – Extra lines to add into the submit file.
- **verbose** (*int*) – Level of logging verbosity.

add_job(*job*)
Add job to Slurm.

Parameters **job** (SlurmJob) – SlurmJob to append to Slurm job list.

build(*makedirs=True*, *fancyname=True*)
Build slurm submit files.

Parameters

- **makedirs** (*bool, optional*) – Create job directories if do not exist.
- **fancyname** (*bool, optional*) – Append the date and unique id number to error, log, output and submit files.

build_submit(*makedirs=True*, *fancyname=True*, *submit_options=None*)
Build and submit sequentially.

Parameters

- **makedirs** (*bool, optional*) – Create directories if not exist.
- **fancyname** (*bool, optional*) – Append date of unique id to submit.
- **submit_options** (*str, optional*) – Options to be passed to 'sbatch'.

Returns **self** – Self object.

Return type object

submit_slurm(*submit_options=None*)
Submit to slurm.

Parameters **submit_options** (*str, optional*) – Submit options appends after sbatch.

visualize(*filename=None*)
Visualize Slurm graph.

Parameters **filename** (*str or None, optional*) – File to save graph diagram to. If `None` then no file is saved. Valid file extensions are 'png', 'pdf', 'dot', 'svg', 'jpeg', 'jpg'.

CHAPTER

FOUR

SLURM

Slurm API

Constructor of `Slurm`:

```
Slurm(name, submit=None, extra_lines=None, verbose=0)
```

`extra_lines` is a list of lines to be added into the submit file. The skeleton of the submit file is as follows:

```
#!/bin/bash
#SBATCH --job-name=<name>
#SBATCH --output=<submit>/<name>.output
#SBATCH --error=<submit>/<name>.error

<extra_lines>

jid0=($(sbatch <dependency if any> <job submit file>))
jid1=($(sbatch <dependency if any> <job submit file>))
...
...
```

CHAPTER

FIVE

SLURMJOB

SlurmJob API

Constructor of `SlurmJob`:

```
SlurmJob(name,
         executable,
         error=None,
         output=None,
         submit=None,
         nodes=None,
         ntasks_per_node=None,
         cpus_per_task=None,
         mem_per_node=None,
         extra_sbatch_options=None,
         extra_srun_options=['ntasks=1', 'exclusive'],
         extra_lines=None,
         modules=None,
         slurm=None,
         arguments=None,
         verbose=0)
```

The constructor provides the basic arguments that are commonly needed to be added into the slurm script.

- `nodes`: Request that a minimum of minnodes nodes be allocated to this job.
- `ntasks_per_node`: Request that ntasks be invoked on each node.
- `cpus_per_task`: Request that ncpus be allocated per process.
- `mem_per_node`: Specify the real memory required per node.
- `modules`: A list of modules to load.

Other arguments can be added through `extra_sbatch_options` and `extra_srun_options`. For more details, please check the documentation of Slurm: sbatch and srun. The skeleton of the submit file is as follows:

```
#!/bin/bash
#SBATCH --job-name=<name>
#SBATCH --output=<submit>/<name>.output
#SBATCH --error=<submit>/<name>.error

```
Speify nodes, ntasks_per_node, cpus_per_task, mem_per_node if provided.
#SBATCH --<nodes,ntasks_per_node,cpus_per_task,mem_per_node>=<>
```
```

(continues on next page)

(continued from previous page)

```
```
Extra sbatch options if extra_sbatch_options if provided.
#SBATCH --<extra_sbatch_option_0>
#SBATCH --<extra_sbatch_option_1>
...
```

```
## Extra lines to be added if extra_lines is provided.
<extra_lines_0>
<extra_lines_1>
...
```

```
## Load modules if modules is provided.
module load <module_0>
module load <module_1>
...
```

srun --<extra_srun_option_0> --<extra_srun_options_1> ... <executable> <argument_0> &
srun --<extra_srun_option_0> --<extra_srun_options_1> ... <executable> <argument_1> &
...
wait
```

CHAPTER

SIX

EXAMPLE

This example walks through a common usage of the package with some mock executables.

Suppose now we want to build a workflow of jobs to analyze some simulated data. The workflow consists of

1) Generate simulated data with the executable `generate-data`.
2) Analyze the simulated data with the executable `analyze-data`.
3) Post-process the analysis results with the executable `postprocess-data`, and at the same time generate plots of the analysis results with the executable `plot`.
4) Summerize the post-processed results and plots in a PDF file with the executable `summary`.

Be reminded that the above are all mock executables which serve for illustration only. You should not try to find those executables in your machine.

Outline of the walk-through:


```
from slurmpter import SlurmJob, Slurm

# Define the error, output and submit directories.
error = "slurm/error"
output = "slurm/output"
submit = "slurm/submit"

# Instantiate a Slurm object which defines the graph of workflow.
slurm = Slurm(name="slurm", submit=submit)

################
# Generate data
```

(continues on next page)

(continued from previous page)

```
################

# Instantiate a SlurmJob object for generating simulated data.
job_gen = SlurmJob(name="generate_data",
                   executable="generate-data",
                   submit=submit,
                   output=output,
                   error=error,
                   slurm=slurm)
# We would like to generate two sets of data.
job_gen.add_arg("--output data_0.txt --input input_0.txt")
job_gen.add_arg("--output data_1.txt --input input_1.txt")

###############
# Analyze data
###############

# Instantiate a SlurmJob object for analyzing the simulated data.
job_analyze = SlurmJob(name="analyze_data",
                       executable="analyze-data",
                       submit=submit,
                       output=output,
                       error=error,
                       slurm=slurm)
# Since we must have the simulated data to be generated before doing the analysis,␣
↪the job_gen is the parent job of job_analyze.
job_analyze.add_parent(job_gen)
# We analyze the two sets of data and get the result outputs.
job_analyze.add_arg("--output result_0.txt --input data_0.txt")
job_analyze.add_arg("--output result_1.txt --input data_1.txt")

########################
# Post-process results
########################

# Instantiate a SlurmJob object for post-processing the analysis results, e.g.␣
↪merging the results etc..
job_postprocess = SlurmJob(name="postprocess_data",
                           executable="postprocess-data",
                           submit=submit,
                           output=output,
                           error=error,
                           slurm=slurm)
# The result outputs have to be ready before we do the post-processing, so the job_
↪analyze is the parent job of job_postprocess.
job_postprocess.add_parent(job_analyze)
# We post-process the result outputs i.e. result_0.txt and result_1.txt.
job_postprocess.add_arg("--output proprocessed_results.txt --input result_0.txt␣
↪result_1.txt")

###################
# Plot the results
###################

# Instantiate a SlurmJob object for plotting the analysis results.
job_plot = SlurmJob(name="plot",
                    executable="plot",
```

(continues on next page)

(continued from previous page)

```
                    submit=submit,
                    output=output,
                    error=error,
                    slurm=slurm)
# Plotting only needs the analysis results, so it can happen concurrently with post-
↪processing.
job_plot.add_parent(job_analyze)
# Generate plots of each of the analysis results.
job_plot.add_arg("--output result_0_plot.pdf --input result_0.txt")
job_plot.add_arg("--output result_1_plot.pdf --input result_1.txt")

#####################
# Generate a summary
#####################

# Instantiate a SlurmJob object for generating summary.
job_summary = SlurmJob(name="summary",
                       executable="generate-summary",
                       submit=submit,
                       output=output,
                       error=error,
                       slurm=slurm)
# The summary file needs the post-processed result and the plots.
job_summary.add_parents([job_postprocess,job_plot])
# Generate the summary as a PDF document.
job_summary.add_arg("--output summary.pdf --input processed_results.txt result_0_plot.
↪pdf result_1_plot.pdf")

# Build the submit files.
slurm.build()
# Call build_submit() if you want to submit the jobs immediately after the build.
# slurm.build_submit()
```

You should see the following submit files in `slurm/submit` if they are successfully built.

`slurm_<date>_01.submit`:

```
#!/bin/bash
#SBATCH --job-name=slurm_<date>_01
#SBATCH --output=slurm/submit/slurm_<date>_01.output
#SBATCH --error=slurm/submit/slurm_<date>_01.error

jid0=($(sbatch slurm/submit/generate_data_<date>_01.submit))
jid1=($(sbatch --dependency=afterok:${jid0[-1]} slurm/submit/analyze_data_<date>_01.
↪submit))
jid2=($(sbatch --dependency=afterok:${jid1[-1]} slurm/submit/postprocess_data_<date>_
↪01.submit))
jid3=($(sbatch --dependency=afterok:${jid1[-1]} slurm/submit/plot_<date>_01.submit))
jid4=($(sbatch --dependency=afterok:${jid2[-1]}:${jid3[-1]} slurm/submit/summary_
↪<date>_01.submit))
```

`generate_data_<date>_01.submit`:

```
#!/bin/bash
#SBATCH --job-name=generate_data_<date>_01
#SBATCH --output=slurm/output/generate_data_<date>_01.output
#SBATCH --error=slurm/error/generate_data_<date>_01.error
```

(continues on next page)

(continued from previous page)

```
srun --ntasks=1 --exclusive generate-data --output data_0.txt --input input_0.txt &
srun --ntasks=1 --exclusive generate-data --output data_1.txt --input input_1.txt &
wait
```

analyze_data_<date>_01.submit:

```
#!/bin/bash
#SBATCH --job-name=analyze_data_<date>_01
#SBATCH --output=slurm/output/analyze_data_<date>_01.output
#SBATCH --error=slurm/error/analyze_data_<date>_01.error

srun --ntasks=1 --exclusive analyze-data --output result_0.txt --input data_0.txt &
srun --ntasks=1 --exclusive analyze-data --output result_1.txt --input data_1.txt &
wait
```

postprocess_data_<date>_01.submit:

```
#!/bin/bash
#SBATCH --job-name=postprocess_data_<date>_01
#SBATCH --output=slurm/output/postprocess_data_<date>_01.output
#SBATCH --error=slurm/error/postprocess_data_<date>_01.error

srun --ntasks=1 --exclusive postprocess-data --output proprocessed_results.txt --
↪input result_0.txt result_1.txt &
wait
```

plot_<date>_01.submit:

```
#!/bin/bash
#SBATCH --job-name=plot_<date>_01
#SBATCH --output=slurm/output/plot_<date>_01.output
#SBATCH --error=slurm/error/plot_<date>_01.error

srun --ntasks=1 --exclusive plot --output result_0_plot.pdf --input result_0.txt &
srun --ntasks=1 --exclusive plot --output result_1_plot.pdf --input result_1.txt &
wait
```

summary_<date>_01.submit:

```
#!/bin/bash
#SBATCH --job-name=summary_<date>_01
#SBATCH --output=slurm/output/summary_<date>_01.output
#SBATCH --error=slurm/error/summary_<date>_01.error

srun --ntasks=1 --exclusive summary --output summary.pdf --input processed_results.
↪txt result_0_plot.pdf result_1_plot.pdf &
wait
```

You can either call `slurm.build_submit()` to squentially build and submit the jobs or call `slurm.submit_slurm()` to submit the jobs after the submit files are built. Alternatively, you can type the following command to submit the jobs after the submit files are built:

```
sbatch slurm_<date>_01.submit
```

6.1 SlurmJob and Slurm objects

```
from slurmpter import SlurmJob, Slurm
```

`Slurm` is a collection of `SlurmJob` which could have different priorities to run using Slurm.

6.2 Job directories

```
error = "slurm/error"
output = "slurm/output"
submit = "slurm/submit"
```

The submit files are built in the `submit` directory, and also the standard output and standard error from `Slurm` are written into the `submit` directory. The standard output and standard error from the `SlurmJob` are written into the `output` and `error` directories respectively.

6.3 Construct a Slurm object

```
# Instantiate a Slurm object which defines the graph of workflow.
slurm = Slurm(name="slurm", submit=submit)
```

You have to first construct a `Slurm` which will hold all the `SlurmJob`. The `name` of the object defines the prefix of the ouput submit file, and therefore you should use different names for all `Slurm` and `SlurmJob` objects.

6.4 Construct a SlurmJob object for generating simulated data

```
################
# Generate data
################

# Instantiate a SlurmJob object for generating simulated data.
job_gen = SlurmJob(name="generate_data",
                   executable="generate-data",
                   submit=submit,
                   output=output,
                   error=error,
                   slurm=slurm)
# We would like to generate two sets of data.
job_gen.add_arg("--output data_0.txt --input input_0.txt")
job_gen.add_arg("--output data_1.txt --input input_1.txt")
```

Usage of `generate-data`:

```
generate-data --output <output> --input <input>``
```

`generate-data` ingests the input file `<input>` and writes the simulated data to `<output>`.

`name` defines the name of the job. `executable` is the name of the executable. You should make sure `executable` can be found in `PATH`. `slurm=slurm` adds this `SlurmJob` object to the `Slurm` object `slurm` that we created above.

Two arguments are added to the job via `job_gen.add_arg()`. The `SlurmJob` then defines two independent runs with the commands

```
generate-data --output data_0.txt --input input_0.txt

generate-data --output data_1.txt --input input_1.txt
```

respectively. The two runs can occur concurrently.

6.5 Construct a SlurmJob object for analyzing data

```
###############
# Analyze data
###############

# Instantiate a SlurmJob object for analyzing the simulated data.
job_analyze = SlurmJob(name="analyze_data",
                       executable="analyze-data",
                       submit=submit,
                       output=output,
                       error=error,
                       slurm=slurm)
# Since we must have the simulated data to be generated before doing the analysis,␣
↪the job_gen is the parent job of job_analyze.
job_analyze.add_parent(job_gen)
# We analyze the two sets of data and get the result outputs.
job_analyze.add_arg("--output result_0.txt --input data_0.txt")
job_analyze.add_arg("--output result_1.txt --input data_1.txt")
```

Usage of `analyze-data`:

```
analyze-data --output <output> --input <input>
```

`analyze-data` ingests the simulated data `<input>` and outputs the analysis result to `<output>`.

Here the `job_analyze` ingests the output files `data_0.txt` and `data_1.txt` from `job_gen` and writes the anlysis results to `` `result_0.txt `` and `result_1.txt` respectively. `job_analyze.add_parent(job_gen)` forces the `job_analyze` to start after `job_gen` has completed all the runs and all exit normally.

6.6 Construct a SlurmJob object for postprocessing

```
######################
# Post-process results
######################

# Instantiate a SlurmJob object for post-processing the analysis results, e.g.␣
↪merging the results etc..
job_postprocess = SlurmJob(name="postprocess_data",
                           executable="postprocess-data",
                           submit=submit,
                           output=output,
                           error=error,
```

(continues on next page)

(continued from previous page)

```
                              slurm=slurm)
# The result outputs have to be ready before we do the post-processing, so the job_
↪analyze is the parent job of job_postprocess.
job_postprocess.add_parent(job_analyze)
# We post-process the result outputs i.e. result_0.txt and result_1.txt.
job_postprocess.add_arg("--output proprocessed_results.txt --input result_0.txt␣
↪result_1.txt")
```

Usage of `postprocess-data`:

```
postprocess-data --output <output> --input <intput>
```

`postprocess-data` ingests the result files `<input>` and writes the processed data to `<output>`.

The job must occur after `job_analyze` completes all the runs to have the result files ready for post-processing.

6.7 Construct a SlurmJob object for plotting result

```
###################
# Plot the results
###################

# Instantiate a SlurmJob object for plotting the analysis results.
job_plot = SlurmJob(name="plot",
                    executable="plot",
                    submit=submit,
                    output=output,
                    error=error,
                    slurm=slurm)
# Plotting only needs the analysis results, so it can happen concurrently with post-
↪processing.
job_plot.add_parent(job_analyze)
# Generate plots of each of the analysis results.
job_plot.add_arg("--output result_0_plot.pdf --input result_0.txt")
job_plot.add_arg("--output result_1_plot.pdf --input result_1.txt")
```

Usage of `plot`:

```
plot --output <output> --input <input>
```

`plot` ingests a result file `<input>` and generates the plot to `<output>`.

Similar to `job_postprocess`, `job_plot` must occur after `job_analyze` to have all the result files ready for plotting.

Notice that the parents of `job_postprocess` and `job_plot` are both `job_analyze`, but `job_postprocess` and `job_plot` do not depend on each other. `job_postprocess` and `job_plot` can occur concurrently after `job_analyze` finishes.

6.8 Construct a SlurmJob object for generating a summary

```
####################
# Generate a summary
####################

# Instantiate a SlurmJob object for generating summary.
job_summary = SlurmJob(name="summary",
                       executable="generate-summary",
                       submit=submit,
                       output=output,
                       error=error,
                       slurm=slurm)
# The summary file needs the post-processed result and the plots.
job_summary.add_parents([job_postprocess,job_plot])
# Generate the summary as a PDF document.
job_summary.add_arg("--output summary.pdf --input processed_results.txt result_0_plot.
↪pdf result_1_plot.pdf")
```

Usage of `summary`:

```
summary --output <output> --input <intput>
```

`summary` ingests the post-processed data and plots `<input>` and writes a summary file to `<output>`.

`job_summary` depends on two jobs i.e. `job_postprocess` and `job_plot`. You can add multiple jobs as the parents at the same time by passing a list of jobs to `add_parents`.

6.9 Build and submit the jobs

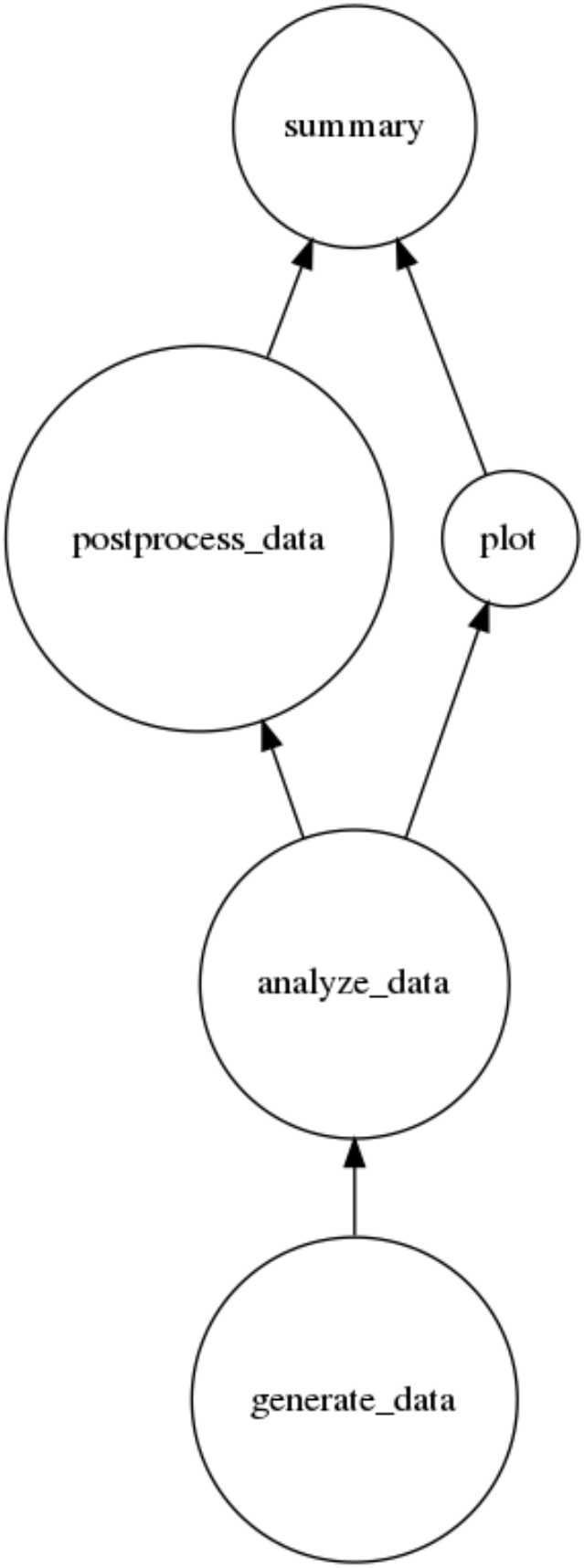
```
# Build the submit files.
slurm.build()
# Call build_submit() if you want to submit the jobs immediately after the build.
# slurm.build_submit()
```

You can either call `build()` to build the submit files but do not submit the jobs yet, or call `build_submit()` to build the submit files and then submit the jobs immediately.

6.10 Visualize the workflow

```
slurm.visualize("workflow.pdf")
```

You can generate a figure of the workflow by calling `visualize()` of the `Slurm` object.


summary
postprocess_data
plot
analyze_data
generate_data

CHAPTER

SEVEN

ALIASES OF PYCONDOR CLASSES

One can also call the `Dagman` and `Job` in PyCondor through the aliases `Dagman` and `DagmanJob` respectively to build HTCondor submit files.

```
from slurmpter import Dagman
from slurmpter import DagmanJob
```

Please visit the PyCondor documentation for more details of the usage.

CHAPTER

EIGHT

CHANGELOG

All notable changes to this project will be documented in this page.

The format is based on Keep a Changelog, and this project adheres to Semantic Versioning.

8.1 [1.0.0] - 2020-10-29

Added

- Added classes `Slurm` and `SlurmJob` which use the `PyCondor` as the backend to handle a workflow of jobs to build and submit Slurm submit files.

CHAPTER

NINE

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

S

INDEX

Symbols

A

B

H

M

S

V