b2luigi Documentation

Release 0.2.1

Nils Braun

Aug 04, 2018

Contents

`b2luigi` - bringing batch 2 luigi!

`b2luigi` is a helper package for `luigi` for scheduling large luigi workflows on a batch system. It is as simple as

```
import b2luigi


class MyTask(b2luigi.Task):
    def output(self):
        return b2luigi.LocalTarget("output_file.txt")

    def run(self):
        with self.output().open("w") as f:
            f.write("This is a test\n")


if __name__ == "__main__":
    b2luigi.process(MyTask(), batch=True)
```

Jump right into it with out *Quick Start*.

If you have never worked with `luigi` before, you may want to have a look into the luigi documentation. But you can learn most of the nice features also from this documentation!

Attention: The API of `b2luigi` is still under construction. Please remember this when using the package in production!

CHAPTER 1

Why not use the already created batch tasks?

Luigi already contains a large set of tasks for scheduling and monitoring batch jobs[1]. But for thousands of tasks in very large projects with different task-defining libraries, you have some problems:

- **You want to run many (like many!) batch jobs in parallel** In other implementations, for every running batch job you also need a running task that monitors it. On most of the systems, the maximal number of processes is limited per user, you will not be able to run more batch jobs than this But what do you do if you have thousands of tasks to do?
- **You have already a large set of luigi tasks in your project** In other implementations you either have to override a `work` function (and you are not allowed to touch the `run` function) or they can only run an external command, which you need to define. The first approach plays not well when mixing non-batch and batch task libraries and the second has problems when you need to pass complex arguments to the external command (via command line).
- **You do not know which batch system you will run on** Currently, the batch tasks are mostly defined for a specific batch system. But what if you want to switch from AWS to Azure? From LSF to SGE?

Entering `b2luigi`, which tries to solve all this (but was heavily inspired by the previous implementations):

- You can run as many tasks as your batch system can handle in parallel! There will only be a single process running on your submission machine.
- No need to rewrite your tasks! Just call them with `b2luigi.process(.., batch=True)` or with `python file.py --batch` and you are ready to go!
- Switching the batch system is just a single change in a config file or one line in python. In the future, there will even be an automatic discovery of the batch system to use.

[1] https://github.com/spotify/luigi/blob/master/luigi/contrib/sge.py

CHAPTER 2

It this the only thing I can do with b2luigi?

As `b2luigi` should help you with large luigi projects, we have also included some helper functionalities for `luigi` tasks and task handling. `b2luigi` task is a super-hero version of `luigi` task, with simpler handling for output and input files. Also, we give you working examples and best-practices for better data management and how to accomplish your goals, that we have learned with time.

CHAPTER 3

Why are you still talking, lets use it!

Have a look into the *Quick Start* or one of the *Advanced Examples*.

You can also start reading the *API Documentation* or the code on github.

If you find any bugs or want to improve the documentation, please send me a pull request.

This project is in beta. Please be extra cautious when using in production mode. You can help me by working with one of the todo items described in *Development and TODOs*.

CHAPTER 4

Content

4.1 Installation

This installation description is for the general user. If you are using the Belle II software, see below:

1. Setup your local environment. For example, run:

```
source venv/bin/activate
```

2. Install b2luigi from pipy into your environment.

   (a) If you have a local installation, you can use the normal setup command

```
pip3 install b2luigi
```

   (b) If this fails because you do not have write access to where your virtual environment lives, you can also install b2luigi locally:

```
pip3 install --user b2luigi
```

   This will automatically also install *luigi* into your current environment. Please make sure to always setup your environment correctly before using *b2luigi*.

Now you can go on with the *Quick Start*.

4.1.1 b2luigi and Belle II

1. Setup your local environment. You can use a local environment (installed on your machine) or a release on cvmfs. For example, run:

```
source /cvmfs/belle.cern.ch/tools/b2setup prerelease-02-00-00c
```

   Or you setup your local installation

```
cd release-directory
source tools-directory/b2setup
```

2. Install b2luigi from pipy into your environment.

   (a) If you have a local installation, you can use the normal setup command

```
pip3 install b2luigi
```

   (b) If you are using an installation from cvmfs, you need to add the `user` flag.

```
pip3 install --user b2luigi
```

The examples in this documentation are all shown with calling `python`, but basf2 users need to use `python3` instead.

Please also have a look into the specific basf2-examples-label.

4.2 Quick Start

We use a very simple task definition file and submit it to a LSF batch system.

Hint: Currently, there is only an implementation for the LSF batch system. More will come soon!

Our task will be very simple: we want to create 100 files with some random number in it. Later, we will build the average of those numbers.

1. Open a code editor and create a new file `simple-example.py` with the following content:

```
1  import b2luigi
2  import random
3
4
5  class MyNumberTask(b2luigi.Task):
6      some_parameter = b2luigi.Parameter()
7
8      def output(self):
9          return b2luigi.LocalTarget(f"results/output_file_{self.some_
   ↪parameter}.txt")
10
11     def run(self):
12         random_number = random.random()
13         with self.output().open("w") as f:
14             f.write(f"{random_number}\n")
15
16
17 if __name__ == "__main__":
18     b2luigi.process([MyNumberTask(some_parameter=i) for i in range(100)])
```

   Each building block in `(b2)luigi` is a `b2luigi.Task`. It defines (which its run function), what should be done. A task can have parameters, as in our case the `some_parameter` defined in line 6. Each task needs to define, what it will output in its `output` function.

   In our run function, we generate a random number and write it to the output file, which is named after the parameter of the task and stored in a result folder.

> **Hint:** For those of you who have already used `luigi` most of this seems familiar. Actually, `b2luigi`'s task is a superset of `luigi`'s, so you can reuse your old scripts! `b2luigi` will not care, which one you are using. But we strongly advice you to use `b2luigi`'s task, as it has some more superior functions (see below).

2. Call the newly created file with python:

```
python simple-example.py --batch
```

   Instead of giving the batch parameter in as argument, you can also add it to the `b2luigi.process(.., batch=True)` call.

   Each task will be scheduled as a batch job to your LSF queue. Using the dependency management of `luigi`, the batch jobs are only scheduled when all dependencies are fulfilled saving you some unneeded CPU time on the batch system. This means although you have requested 200 workers, you only need 100 workers to fulfill the tasks, so only 100 batch jobs will be started. On your local machine runs only the scheduling mechanism needing only a small amount of CPUs.

   > **Hint:** If you have no LSF queue ready, you can also remove the *batch* argument. This will fall back to a normal `luigi` execution.

3. After the job is completed, you will see something like:

```
===== Luigi Execution Summary =====

Scheduled 100 tasks of which:
* 100 ran successfully:
    - 100 MyTask(some_parameter=0,1,10,11,12,13,14,15,16,17,18,...)

This progress looks :) because there were no failed tasks or missing dependencies

===== Luigi Execution Summary =====
```

   The log files for each task are written to the *logs* folder.

   After a job is submitted, `b2luigi` will check if it is still running or not and handle failed or done tasks correctly.

4. The defined outputs will in most of the cases depend on the parameters of the task, as you do not want to override your files from different tasks. The cumbersome work of keeping track of the correct outputs can be handled by `b2luigi`, which will also help you ordering your files at no cost. This is especially useful in larger projects, when many people are defining and executing tasks.

   This code listing shows the same task, but this time written using the helper functions given by `b2luigi`.

```
1  import b2luigi
2  import random
3
4
5  class MyNumberTask(b2luigi.Task):
6      some_parameter = b2luigi.Parameter()
7
8      def output(self):
9          yield self.add_to_output("output_file.txt")
10
11     def run(self):
```

(continues on next page)

(continued from previous page)

```
        random_number = random.random()

        with open(self.get_output_file_name("output_file.txt"), "w") as f:
            f.write(f"{random_number}\n")


if __name__ == "__main__":
    b2luigi.process([MyNumberTask(some_parameter=i) for i in range(100)])
```

Before you execute the file (e.g. with `--batch`), add a `settings.json` with the following content in your current working directory:

```
{
        "result_path": "results"
}
```

If you now call

```
python simple-example.py --batch
```

you are basically doing the same as before, with some very nice benefits:

- The parameter values are automatically added to the output file (have a look into the *results/* folder to see how it works)
- The `settings.json` will be used by all tasks in this folder and in each sub-folder. You can use it to define project settings (like result folders) and specific settings for your local sub project. Read the documentation on `b2luigi.get_setting()` for more information on how to use it.

5. Let's add some more tasks to our little example. We want to use the currently created files and add them all together to an average number. So edit your example file to include the following content:

```
import b2luigi
import random


class MyNumberTask(b2luigi.Task):
    some_parameter = b2luigi.Parameter()

    def output(self):
        yield self.add_to_output("output_file.txt")

    def run(self):
        random_number = random.random()

        with open(self.get_output_file_name("output_file.txt"), "w") as f:
            f.write(f"{random_number}\n")


class MyAverageTask(b2luigi.Task):
    def requires(self):
        for i in range(100):
            yield self.clone(MyNumberTask, some_parameter=i)

    def output(self):
        yield self.add_to_output("average.txt")

```

(continues on next page)

(continued from previous page)

```
26      def run(self):
27          # Build the mean
28          summed_numbers = 0
29          counter = 0
30          for input_file in self.get_input_file_names("output_file.txt"):
31              with open(input_file, "r") as f:
32                  summed_numbers += float(f.read())
33                  counter += 1
34
35          average = summed_numbers / counter
36
37          with open(self.get_output_file_name("average.txt"), "w") as f:
38              f.write(f"{average}\n")
39
40
41  if __name__ == "__main__":
42      b2luigi.process(MyAverageTask(), workers=200)
```

See how we defined dependencies in line 19 with the `requires` function. By calling `clone` we make sure that any parameters from the current task (which are none in our case) are copied to the dependencies.

Hint: Again, expert `luigi` users will not see anything new here.

By using the helper functions `b2luigi.Task.get_input_file_names()` and `b2luigi.Task.get_output_file()` the output file name generation with parameters is transparent to you as a user. Super easy!

When you run the script, you will see that `luigi` detects your already run files from before (the random numbers) and will not run the task again! It will only output a file in *results/average.txt* with a number near 0.5.

You are now ready to face some more *Advanced Examples* or have a look into the *FAQ*.

4.2.1 Choosing the LSF queue

By default, all tasks will be sent to the short queue. This behaviour can be changed on a per task level by giving the task a property called `queue` and setting it to the queue it should run on, e.g.

```
class MyLongTask(b2luigi.Task):
    queue = "l"
```

4.2.2 Start a Central Scheduler

When the number of tasks grows, it is sometimes hard to keep track of all of them (despite the summary in the end). For this, `luigi` brings a nice visualisation tool called the central scheduler.

To start this you need to call the `luigid` executable. Where to find this depends on your installation type:

1. If you have a installed `b2luigi` without user flag, you can just call the executable as it is already in your path:

```
luigid --port PORT
```

2. If you have a local installation, luigid is installed into your home directory:

```
~/.local/bin/luigid --port PORT
```

The default port is 8082, but you can choose any non-occupied port.

The central scheduler will register the tasks you want to process and keep track of which tasks are already done.

To use this scheduler, call `b2luigi` by giving the connection details:

```
python simple-task.py [--batch] --scheduler-host HOST --scheduler-port PORT
```

which works for batch as well as non-batch jobs. You can now visit the url http://HOST:PORT with your browser and see a nice summary of the current progress of your tasks.

4.2.3 Drawbacks of the batch mode

Although the batch mode has many benefits, it would be unfair to not mention its downsides:

- We are currently assuming that you have the same environment setup on the batch system as locally (actually, we are copying the console environment variables) and we will call the python executable which runs your scheduling job.
- You have to choose the queue depending in your requirements (e.g. wall clock time) by yourself. So you need to make sure that the tasks will actually finish before the batch system kills them because of timeout.
- There is currently now resubmission implemented. This means dying jobs because of batch system failures are just dead. But because of the dependency checking mechanism of `luigi` it is simple to just redo the calculation and re-calculate what is missing.
- The `luigi` feature to request new dependencies while task running (via yield) is not implemented for the batch mode.

4.3 Advanced Examples

4.4 Basf2 specific examples

The following examples are not of interest to the general audience, but only for basf2 users.

4.4.1 Standard Simulation, Reconstruction and some nTuple Generation

```
import b2luigi as luigi
from b2luigi.basf2_helper import Basf2PathTask, Basf2nTupleMergeTask

from enum import Enum

import basf2

import modularAnalysis
import simulation
import generators
import reconstruction
from ROOT import Belle2
```

(continues on next page)

(continued from previous page)

```
class SimulationType(Enum):
    y4s = "Y(4S)"
    continuum = "Continuum"


class SimulationTask(Basf2PathTask):
    n_events = luigi.IntParameter()
    event_type = luigi.EnumParameter(enum=SimulationType)

    def create_path(self):
        path = basf2.create_path()
        modularAnalysis.setupEventInfo(self.n_events, path)

        if self.event_type == SimulationType.y4s:
            dec_file = Belle2.FileSystem.findFile('analysis/examples/tutorials/B2A101-
↪Y4SEventGeneration.dec')
        elif self.event_type == SimulationType.continuum:
            dec_file = Belle2.FileSystem.findFile('analysis/examples/tutorials/B2A102-
↪ccbarEventGeneration.dec')
        else:
            raise ValueError(f"Event type {self.event_type} is not valid. It should␣
↪be either 'Y(4S)' or 'Continuum'!")

        generators.add_evtgen_generator(path, 'signal', dec_file)
        modularAnalysis.loadGearbox(path)
        simulation.add_simulation(path)

        path.add_module('RootOutput', outputFileName=self.get_output_file_name(
↪'simulation_full_output.root'))

        return path

    def output(self):
        yield self.add_to_output("simulation_full_output.root")


@luigi.requires(SimulationTask)
class ReconstructionTask(Basf2PathTask):
    def create_path(self):
        path = basf2.create_path()

        path.add_module('RootInput', inputFileNames=self.get_input_file_names(
↪"simulation_full_output.root"))
        modularAnalysis.loadGearbox(path)
        reconstruction.add_reconstruction(path)

        modularAnalysis.outputMdst(self.get_output_file_name("reconstructed_output.
↪root"), path=path)

        return path

    def output(self):
        yield self.add_to_output("reconstructed_output.root")


@luigi.requires(ReconstructionTask)
class AnalysisTask(Basf2PathTask):
```

(continues on next page)

(continued from previous page)

```
    def create_path(self):
        path = basf2.create_path()
        modularAnalysis.inputMdstList('default', self.get_input_file_names(
↪"reconstructed_output.root"), path=path)
        modularAnalysis.fillParticleLists([('K+', 'kaonID > 0.1'), ('pi+', 'pionID >␣
↪0.1')], path=path)
        modularAnalysis.reconstructDecay('D0 -> K- pi+', '1.7 < M < 1.9', path=path)
        modularAnalysis.fitVertex('D0', 0.1, path=path)
        modularAnalysis.matchMCTruth('D0', path=path)
        modularAnalysis.reconstructDecay('B- -> D0 pi-', '5.2 < Mbc < 5.3', path=path)
        modularAnalysis.fitVertex('B+', 0.1, path=path)
        modularAnalysis.matchMCTruth('B-', path=path)
        modularAnalysis.variablesToNtuple('D0',
                                          ['M', 'p', 'E', 'useCMSFrame(p)',
↪'useCMSFrame(E)',
                                           'daughter(0, kaonID)', 'daughter(1, pionID)
↪', 'isSignal', 'mcErrors'],
                                          filename=self.get_output_file_name("D_n_
↪tuple.root"),
                                          path=path)
        modularAnalysis.variablesToNtuple('B-',
                                          ['Mbc', 'deltaE', 'isSignal', 'mcErrors', 'M
↪'],
                                          filename=self.get_output_file_name("B_n_
↪tuple.root"),
                                          path=path)
        return path

    def output(self):
        yield self.add_to_output("D_n_tuple.root")
        yield self.add_to_output("B_n_tuple.root")


class MasterTask(Basf2nTupleMergeTask):
    n_events = luigi.IntParameter()

    def requires(self):
        for event_type in SimulationType:
            yield self.clone(AnalysisTask, event_type=event_type)


if __name__ == "__main__":
    luigi.process(MasterTask(n_events=1), workers=4)
```

4.5 API Documentation

`b2luigi` summarizes different topics to help you in your everyday task creation and processing. Most important is the `b2luigi.process()` function, which lets you run arbitrary task graphs on the batch. It is very similar to `luigi.build`, but lets you hand in additional parameters for steering the batch execution.

4.5.1 Top-Level Function

4.5.2 Super-hero Task Classes

If you want to use the default `luigi.Task` class or any derivative of it, you are totally fine. No need to change any of your scripts! But if you want to take advantage of some of the recipies we have developed to work with large luigi task sets, you can use the drop in replacements from the `b2luigi` package. All task classes (except the `b2luigi.DispatchableTask`) are superclasses of a `luigi` class. As we import `luigi` into `b2luigi`, you just need to replace

```
import luigi
```

with

```
import b2luigi as luigi
```

and you will have all the functionality of `luigi` and `b2luigi` without the need to change anything!

4.5.3 Settings

4.5.4 Other functions

b2luigi.core.utils module

b2luigi.batch package

b2luigi.basf2_helper package

b2luigi.basf2_helper.data module

b2luigi.basf2_helper.targets module

b2luigi.basf2_helper.tasks module

b2luigi.basf2_helper.utils module

4.6 Run Modes

The run mode can be chosen by calling your python file with

```
python file.py --mode
```

or by calling `b2luigi.process` with a given mode set to `True`

```
b2luigi.process(.., mode=True)
```

where mode can be one of:

- **batch**: Run the tasks on a batch system, as described in *Quick Start*. The maximal number of batch jobs to run in parallel (jobs in flight) is equal to the number of workers. This is 1 by default, so you probably want to change this. By default, LSF is used as a batch system. If you want to change this, set the corresponding `batch_system` setting-label to one of the supported systems.

- **dry-run**: Similar to the dry-run funtionality of `luigi`, this will not start any tasks but just tell you, which tasks it would run. The exit code is 1 in case a task needs to run and 0 otherwise.
- **show-output**: List all output files that this has produced/will produce. Files which already exist (where the targets define, what exists mean in this case) are marked as green whereas missing targets are marked red.
- **test**: Run the tasks normally (no batch submission), but turn on debug logging of `luigi`. Also, do not dispatch any task (if requested) and print the output to the console instead of in log files.

Additional console arguments:

- **–scheduler-host** and **–scheduler-port**: If you have set up a central scheduler, you can pass this information here easily. This works for batch or non-batch submission but is turned of for the test mode.

4.7 FAQ

TODO

4.8 Development and TODOs

You want to help developing `b2luigi`? Great! Have your github account ready and let's go!

4.8.1 Local Development

You want to help developing `b2luigi`? Great! Here are some first steps to help you dive in:

1. Make sure you uninstall `b2luigi` if you have installed if from pypi

   ```
   pip3 uninstall b2luigi
   ```

2. Clone the repository from github

   ```
   git clone https://github.com/nils-braun/b2luigi
   ```

3. `b2luigi` is not using `setuptools` but the newer (and better) flit as a a builder. Install it via

   ```
   pip3 [ --user ] install flit
   ```

   You can now install `b2luigi` from the cloned git repository in development mode:

   ```
   flit install -s
   ```

4. The documentation is hosted on read the docs and build automatically on every commit to master. You can (and should) also build the documentation locally by installing `sphinx`

   ```
   pip3 [ --user ] install sphinx sphinx-autobuild
   ```

   And starting the automatic build process in the projects root folder

   ```
   sphinx-autobuild docs build
   ```

   The autobuild will rebuild the project whenever you change something. It displays a URL where to find the created docs now (most likely http://127.0.0.1:8000). Please make sure the documentation looks fine before creating a pull request.

5. If you are a core developer and want to release a new version:
   (a) Make sure all changes are committed and merged on master
   (b) Use the `bumpversion` package to update the version in the python file `b2luigi/__init__.py` as well as the git tag. `flit` will automatically use this.

```
bumpversion patch/minor/major
```

   (c) Push the new commit and the tags

```
git push
git push --tags
```

   (d) Publish to pipy

```
flit publish
```

   At a later stage, I will try to automate this.

4.8.2 Open TODOs

- Add support for different batch systems, e.g. htcondor and a batch system discovery
- Integrate dirac or other grid systems as another batch system
- Add helper messages on events (e.g. failed)

CHAPTER 5

The name

`b2luigi` stands for multiple things at the same time:

- It brings **b**atch to (**2**) luigi.
- It helps you with the **b**read and **b**utter work in luigi (e.g. proper data management)
- It was developed for the Belle II experiment.

CHAPTER 6

The team

Main developer:

- Nils Braun (nils-braun)

Useful help and testing:

- Felix Metzner
- Patrick Ecker
- Jochen Gemmler

Stolen ideas:

- Implementation of SGE batch system (sge).
- Implementation of LSF batch system (lsf).