b2luigi Documentation

Release 0.3.2

Nils Braun

Jul 04, 2019

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:

a. 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
```

b. 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

b2luigi.**process**(*task_like_elements*, *show_output=False*, *dry_run=False*, *test=False*, *batch=False*, ***kwargs*)

Call this function in your main method to tell b2luigi where your entry point of the task graph is. It is very similar to luigi.build with some additional configuration options.

Example

This example defines a simple task and tells b2luigi to execute it 100 times with different parametes:

```
import b2luigi
import random


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

    def output(self):
        return b2luigi.LocalTarget(f"results/output_file_{self.some_parameter}.txt
↪")

    def run(self):
        random_number = random.random()
        with self.output().open("w") as f:
            f.write(f"{random_number}\n")

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

All flag arguments can also be given as command line arguments. This means the call with:

```
b2luigi.process(tasks, batch=True)
```

is equivalent to calling the script with:

```
python script.py --batch
```

Parameters

- **task_like_elements** (*Task* or list) – Task(s) to execute with luigi. Can either be a list of tasks or a task instance.
- **show_output** (*bool, optional*) – Instead of running the task(s), write out all output files which will be generated marked in color, if they are present already. Good for testing of your tasks will do, what you think they should.
- **dry_run** (*bool, optional*) – Instead od running the task(s), write out which tasks will be executed. This is a simplified form of dependency resolution, so this information may be wrong in some corner cases. Also good for testing.
- **test** (*bool, optional*) – Does neither run on the batch system, with multiprocessing or dispatched (see *DispatchableTask*) but directly on the machine for debugging reasons. Does output all logs to the console.
- **batch** (*bool, optional*) – Execute the tasks on the selected batch system. Refer to *Quick Start* for more information. The default batch system is LSF, but this can be changed with the *batch_system* settings. See *get_setting* on how to define settings.

- ****kwargs** – Additional keyword arguments passed to `luigi.build`.

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!

class `b2luigi.`**Task**(**args*, ***kwargs*)

Bases: `luigi.task.Task`

Drop in replacement for `luigi.Task` which is 100% API compatible. It just adds some useful methods for handling output file name generation using the parameters of the task. See *Quick Start* on information on how to use the methods.

Example:

```
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")

    def run(self):
        # Build the mean
        summed_numbers = 0
        counter = 0
        for input_file in self.get_input_file_names("output_file.txt"):
            with open(input_file, "r") as f:
                summed_numbers += float(f.read())
                counter += 1

        average = summed_numbers / counter

        with self.get_output_file("average.txt").open("w") as f:
            f.write(f"{average}\n")
```

add_to_output(*output_file_name*)

Call this in your output() function to add a target to the list of files, this task will output. Always use in combination with *yield*. This function will automatically add all current parameter values to the file name when used in the form `result_path/param_1=value/param_2=value/output_file_name`

This function will automatically use a `LocalTarget`. If you do not want this, you can override the `_get_output_file_target` function.

Example

This adds two files called `some_file.txt` and `some_other_file.txt` to the output:

```
def output(self):
    yield self.add_to_output("some_file.txt")
    yield self.add_to_output("some_other_file.txt")
```

Parameters **output_file_name** (`str`) – the file name of the output file. Refer to this file name as a key when using *get_input_file_names*, `get_output_file_names` or `get_output_file`.

get_input_file_names(*key=None*)

Get a dictionary of input file names of the tasks, which are defined in our requirements. Either use the key argument or dictionary indexing with the key given to *add_to_output* to get back a list (!) of file paths.

Parameters **key** (`str`, optional) – If given, only return a list of file paths with this given key.

Returns If key is none, returns a dictionary of keys to list of file paths. Else, returns only the list of file paths for this given key.

get_output_file_name(*key*)

Analogous to *get_input_file_names* this function returns a an output file defined in out output function with the given key.

In contrast to *get_input_file_names*, only a single file name will be returned (as there can only be a single output file with a given name).

Parameters **key** (`str`) – Return the file path with this given key.

Returns Returns only the file path for this given key.

class b2luigi.**ExternalTask**(**args*, ***kwargs*)

Bases: `b2luigi.core.task.Task`, `luigi.task.ExternalTask`

Direct copy of `luigi.ExternalTask`, but with the capabilities of *Task* added.

class b2luigi.**WrapperTask**(**args*, ***kwargs*)

Bases: `b2luigi.core.task.Task`, `luigi.task.WrapperTask`

Direct copy of `luigi.WrapperTask`, but with the capabilities of *Task* added.

b2luigi.**dispatch**(*run_function*)

In cases you have a run function calling external, probably insecure functionalities, use this function wrapper around your run function.

Example

The run function can include any code you want. When the task runs, it is started in a subprocess and monitored by the parent process. When it dies unexpectedly (e.g. because of a segfault etc.) the task will be marked as failed. If not, it is successful. The log output will be written to two files in the log folder (marked with the parameters of the task), which you can check afterwards:

```
import b2luigi


class MyTask(b2luigi.Task):
```

(continues on next page)

(continued from previous page)

```
    @b2luigi.dispatch
    def run(self):
        call_some_evil_function()
```

Implementation note: In the subprocess we are calling the current sys.executable (which should by python hopefully) with the current input file as a parameter, but let it only run this specific task (by handing over the task id and the *–batch-worker* option). The run function notices this and actually runs the task instead of dispatching again.

You have the possibility to control what exactly is used as executable by setting the "executable" setting, which needs to be a list of strings. Additionally, you can add a `cmd_prefix` parameter to your class, which also needs to be a list of strings, which are prefixed to the current command (e.g. if you want to add a profiler to all your tasks)

class b2luigi.**DispatchableTask**(**args*, ***kwargs*)

Bases: b2luigi.core.task.Task

Instead of using the *dispatch* function wrapper, you can also inherit from this class. Except that, it has exactly the same functionality as a normal *Task*.

Important: You need to overload the process function instead of the run function in this case!

process()

Override this method with your normal run function. Do not touch the run function itself!

4.5.3 Parameters

As `b2luigi` automatically also imports `luigi`, you can use all the parameters from `luigi` you know and love. We have just added a single new flag called `hashed` to the parameters constructor. Turning it to true (it is turned off by default) will make `b2luigi` use a hashed version of the parameters value, when constructing output or log file paths. This is especially useful if you have parameters, which may include "dangerous" characters, like "/" or "{" (e.g. when using list or dictionary parameters). See also one of our *FAQ*.

4.5.4 Settings

b2luigi.**get_setting**(*key*, *default=None*)

`b2luigi` adds a settings management to `luigi` and also uses it at various places.

With this function, you can get the current value of a specific setting with the given key. If there is no setting defined with this name, either the default is returned or, if you did not supply any default, a value error is raised.

For information on how settings are set, please see *set_setting*. Settings can be of any type, but are mostly strings.

Parameters

- **key** (str) – The name of the parameter to query.
- **default** (*optional*) – If there is no setting which the name, either return this default or if it is not set, raise a ValueError.

b2luigi.**set_setting**(*key*, *value*)

There are two possibilities to set a setting with a given name:

- Either you use this function and supply the key and the value. The setting is then defined globally for all following calls to *get_setting* with the specific key.
- Or you add a file called `settings.json` the the current working directory *or any folder above that*. In the json file, you need to supply a key and a value for each setting you want to have, e.g:

```
{
    "result_path": "results",
    "some_setting": "some_value"
}
```

By looking also in the parent folders for setting files, you can define project settings in a top folder and specific settings further down in your local folders.

b2luigi.**clear_setting**(*key*)
Clear the setting with the given key

4.5.5 Other functions

b2luigi.**on_temporary_files**(*run_function*)
Wrapper for decorating a task's run function to use temporary files as outputs.

A common problem when using long running tasks in luigi is the thanksgiving bug. It occurs, when you define an output of a task and in its run function you create this output and fill it with content during a long lasting calculation. It may happen, that during the creation of the output and the finish of the calculation some other tasks look if the output is already there, find it existing and assume, that the task is already finished (although there is probably only nonsense in the file).

A solution is already given by luigi itself, when using the temporary_path() function of the file system targets, which is really nice! Unfortunately, this means you have to open all your output files with a context manager and this is very hard to do if you have external tasks also (because they will probably use the output file directly instead of the temporary file version of if).

This wrapper simplifies the usage of the temporary files:

```
import b2luigi

class MyTask(b2luigi.Task):
    def output(self):
        yield self.add_to_output("test.txt")

    @b2luigi.on_temporary_files
    def run(self):
        with open(self.get_output_file_name("test.txt"), "w") as f:
            raise ValueError()
            f.write("Test")
```

Instead of creating the file "test.txt" at the beginning and filling it with content later (which will never happen because of the exception thrown, mich makes the file existing but the task actually not finished), the file will be written to a temporary file first and copied to its final location at the end of the run function (but only if there was no error).

Attention:

The decorator only edits the function get_output_file_name. If you are using the output directly, you have to take care of using the temporary path correctly by yourself!

b2luigi.core.utils module

b2luigi.core.utils.**add_on_failure_function**(*task*)

b2luigi.core.utils.**create_cmd_from_task**(*task*)

b2luigi.core.utils.**create_output_dirs**(*task*)
Create all output dicts if needed. Normally only used internally.

b2luigi.core.utils.**create_output_file_name**(*task*, *base_filename*, *create_folder=False*, *result_path=None*)

b2luigi.core.utils.**fill_kwargs_with_lists**(***kwargs*)
Return the kwargs with each value mapped to [value] if not a list already.

Example: .. code-block:: python

```
>>> fill_kwargs_with_lists(arg_1=[1, 2], arg_2=3)
{'arg_1': [1, 2], 'arg_2': [3]}
```

Parameters **kwargs** – The input keyword arguments

Returns Same as kwargs, but each value mapped to a list if not a list already

b2luigi.core.utils.**filter_from_params**(*output_files*, ***kwargs*)

b2luigi.core.utils.**flatten_to_dict**(*inputs*)
Return a whatever input structure into a dictionary. If it is a dict already, return this. If is is an iterable of dict or dict-like objects, return the merged dictionary. All non-dict values will be turned into a dictionary with value -> {value: value}

Example: .. code-block:: python

```
>>> flatten_to_dict([{"a": 1, "b": 2}, {"c": 3}, "d"])
{'a': 1, 'b': 2, 'c': 3, 'd': 'd'}
```

Parameters **inputs** – The input structure

Returns A dict constructed as described above.

b2luigi.core.utils.**flatten_to_file_paths**(*inputs*)
Take in a structure of something and replace each luigi target by its corresponding path. For dicts, it will replace the value as well as the key. The key will however only by the basename of the path.

Parameters **inputs** – A dict or a luigi target

Returns A dict with the keys replaced by the basename of the targets and the values by the full path

b2luigi.core.utils.**flatten_to_list_of_dicts**(*inputs*)

b2luigi.core.utils.**get_all_output_files_in_tree**(*root_module*, *key=None*)

b2luigi.core.utils.**get_filled_params**(*task*)
Helper function for getting the parameter list with each parameter set to its current value

b2luigi.core.utils.**get_log_file_dir**(*task*)

b2luigi.core.utils.**get_serialized_parameters**(*task*)
Get a string-typed ordered dict of key=value for the significant parameters

b2luigi.core.utils.**get_task_from_file**(*file_name*, *task_name*, ***kwargs*)

b2luigi.core.utils.**on_failure**(*self*, *exception*)

b2luigi.core.utils.**product_dict**(***kwargs*)

Cross-product the given parameters and return a list of dictionaries.

Example: .. code-block:: python

```
>>> list(product_dict(arg_1=[1, 2], arg_2=[3, 4]))
[{'arg_1': 1, 'arg_2': 3}, {'arg_1': 1, 'arg_2': 4}, {'arg_1': 2, 'arg_2': 3}, {
↪'arg_1': 2, 'arg_2': 4}]
```

The thus produced list can directly be used as inputs for a required tasks:

Parameters **kwargs** – Each keyword argument should be an iterable

Returns A list of kwargs where each list of input keyword arguments is cross-multiplied with every other.

b2luigi.core.utils.**remember_cwd**()

Helper contextmanager to stay in the same cwd

b2luigi.core.utils.**task_iterator**(*task*, *only_non_complete=False*)

b2luigi.batch package

class b2luigi.batch.processes.**BatchProcess**(*task*, *scheduler*, *result_queue*, *worker_timeout*)

This is the base class for all batch algorithms that allow luigi to run on a specific batch system. This is an abstract base class and inheriting classes need to supply functionalities for * starting a job using the commands in self.task_cmd * getting the job status of a running, finished or failed job * and killing a job All those commands are called from the main process, which is not running on the batch system. Every batch system that is capable of these functions can in principle work together with b2luigi.

Implementation note: In principle, using the batch system is transparent to the user. In case of problems, it may however be useful to understand how it is working.

When you start your luigi dependency tree with `process(..., batch=True)`, the normal luigi process is started looking for unfinished tasks and running them etc. Normally, luigi creates a process for each running task and runs them either directly or on a different core (if you have enabled more than one worker). In the batch case, this process is not a normal python multiprocessing process, but this BatchProcess, which has the same interface (one can check the status of the process, start or kill it). The process does not need to wait for the batch job to finish but is asked repeatedly for the job status. By this, most of the core functionality of luigi is kept and reused. This also means, that every batch job only includes a single task and is finished whenever this task is done decreasing the batch runtime. You will need exectly as many batch jobs as you have tasks and no batch job will idle waiting for input data as all are scheduled only when the task they should run is actually runnable (the input files are there).

What is the batch command now? In each job, we start the current python interpreter (the one you used to call this main file) to start the very same file again. However, we give it an additional parameter, which tells it to only run one single task. Task can be identified by their task id. A typical task command may look like:

```
/your-path/venv/bin/python /your-project/some-file.py --batch-runner --task-
↪id MyTask_38dsf879w3
```

if you are using a virtual environment and the batch job should run the MyTask. The implementation of the abstract functions is responsible for starting a job with exactly this command (which is stored in self.task_cmd) and writing the log of the job into appropriate locations.

Warning: There are a few drawbacks when using the batch system, which you may have to keep in mind:

- We are currently assuming that you have the same environment setup on the batch system as locally and we will call the python executable which runs your scheduling job. The currently integrated batch processes (e.g. LSF) use the current environment variables including the cwd and PATH also on the batch.
- The `luigi` feature to request new dependencies while task running (via yield) is not implemented for the batch mode.
- Other drawbacks may come from the implemented batch processes.

get_job_status()
Implement this function to return the current job status. How you identify exactly your job is dependent on the implementation and needs to be handled by your own child class.

Must return one item of the JobStatus enumeration: running, aborted, successful. Will only be called after the job is started already but may also be called when the job is finished already. If the task status is unknown, return aborted. If the task has not started already (but is scheduled), return running nevertheless. No matter if aborted via a call to kill_job, by the batch system or by an exception in the job itself, you should return aborted if the job is not finished successfully.

kill_job()
This command is used to abort a job started by the start_job function. It is only called once to abort a job, so make sure to either block until the job is really gone or be sure that it will go down soon. Especially, do not wait until the job is finished. It is called for example when the user presses Ctrl-C.

In some strange corner cases it may happen that this function is called even before the job is started (the start_job function is called). In this case, you do not need to do anything (but not raise an exception).

start_job()
Override this function in your child class to start a job on the batch system. It is called exactly once. You need to store any information identifying your batch job on your own.

After the start_job function is called by the framework (and no exception is thrown), it is assumed that a batch job running exactly the command in self.task_cmd is started or scheduled. Please make sure to have a proper environment in the batch job (e.g. by copying the current environment) and the same python executable (or a similar one) running in the same working directory.

After the job is finished (no matter if aborted or successful) we assume the stdout and stderr is written into the two files given by b2luigi.core.utils.get_log_files(self.task).

class b2luigi.batch.processes.lsf.**LSFProcess**(**args*, ***kwargs*)
Bases: *b2luigi.batch.processes.BatchProcess*

Reference implementation of the batch process for an LSF batch system.

We assume that the batch system shares a file system with the submission node you are currently working on (or at least the current folder is also available there with the same path). We also assume that we can run the same python interpreter there by just copying the current environment and calling it from the same path. Both requirements are fulfilled by a "normal" LSF setup, so you do not keep those in mind typically.

b2luigi.basf2_helper package

b2luigi.basf2_helper.data module

class b2luigi.basf2_helper.data.**CdstDataTask**(**args*, ***kwargs*)
Bases: *b2luigi.basf2_helper.data.DstDataTask*

data_mode = 'cdst'

output()
The output that this Task produces.

The output of the Task determines if the Task needs to be run–the task is considered finished iff the outputs all exist. Subclasses should override this method to return a single `Target` or a list of `Target` instances.

Implementation note If running multiple workers, the output must be a resource that is accessible by all workers, such as a DFS or database. Otherwise, workers might compute the same output since they don't see the work done by other workers.

See Task.output

class b2luigi.basf2_helper.data.**DataMode**
Bases: `enum.Enum`

An enumeration.

cdst = 'cdst'

mdst = 'mdst'

raw = 'raw'

skimmed_raw = 'skimmed_raw'

class b2luigi.basf2_helper.data.**DataTask**(**args*, ***kwargs*)
Bases: `b2luigi.core.task.ExternalTask`

data_mode = <luigi.parameter.EnumParameter object>

experiment_number = <luigi.parameter.IntParameter object>

file_name = <luigi.parameter.Parameter object>

prefix = <luigi.parameter.Parameter object>

run_number = <luigi.parameter.IntParameter object>

class b2luigi.basf2_helper.data.**DstDataTask**(**args*, ***kwargs*)
Bases: *b2luigi.basf2_helper.data.DataTask*

database = <luigi.parameter.IntParameter object>

output()
The output that this Task produces.

The output of the Task determines if the Task needs to be run–the task is considered finished iff the outputs all exist. Subclasses should override this method to return a single `Target` or a list of `Target` instances.

Implementation note If running multiple workers, the output must be a resource that is accessible by all workers, such as a DFS or database. Otherwise, workers might compute the same output since they don't see the work done by other workers.

See Task.output

prod = <luigi.parameter.IntParameter object>

release = <luigi.parameter.Parameter object>

class b2luigi.basf2_helper.data.**MdstDataTask**(**args*, ***kwargs*)
Bases: *b2luigi.basf2_helper.data.DstDataTask*

data_mode = 'mdst'

output()
The output that this Task produces.

The output of the Task determines if the Task needs to be run–the task is considered finished iff the outputs all exist. Subclasses should override this method to return a single `Target` or a list of `Target` instances.

Implementation note If running multiple workers, the output must be a resource that is accessible by all workers, such as a DFS or database. Otherwise, workers might compute the same output since they don't see the work done by other workers.

See Task.output

class b2luigi.basf2_helper.data.**RawDataTask**(**args*, ***kwargs*)
Bases: *b2luigi.basf2_helper.data.DataTask*

data_mode = 'raw'

output()
The output that this Task produces.

The output of the Task determines if the Task needs to be run–the task is considered finished iff the outputs all exist. Subclasses should override this method to return a single `Target` or a list of `Target` instances.

Implementation note If running multiple workers, the output must be a resource that is accessible by all workers, such as a DFS or database. Otherwise, workers might compute the same output since they don't see the work done by other workers.

See Task.output

class b2luigi.basf2_helper.data.**SkimmedRawDataTask**(**args*, ***kwargs*)
Bases: *b2luigi.basf2_helper.data.DstDataTask*

data_mode = 'skimmed_raw'

output()
The output that this Task produces.

The output of the Task determines if the Task needs to be run–the task is considered finished iff the outputs all exist. Subclasses should override this method to return a single `Target` or a list of `Target` instances.

Implementation note If running multiple workers, the output must be a resource that is accessible by all workers, such as a DFS or database. Otherwise, workers might compute the same output since they don't see the work done by other workers.

See Task.output

b2luigi.basf2_helper.data.**clone_on_cdst**(*self*, *task_class*, *experiment_number*, *run_number*, *release*, *prod*, *database*, *prefix=None*, *file_name=None*, ***additional_kwargs*)

b2luigi.basf2_helper.data.**clone_on_mdst**(*self*, *task_class*, *experiment_number*, *run_number*, *release*, *prod*, *database*, *prefix=None*, *file_name=None*, ***additional_kwargs*)

b2luigi.basf2_helper.data.**clone_on_raw**(*self*, *task_class*, *experiment_number*, *run_number*, *prefix=None*, *file_name=None*, ***additional_kwargs*)

b2luigi.basf2_helper.data.**clone_on_skimmed_raw**(*self*, *task_class*, *experiment_number*, *run_number*, *release*, *prod*, *database*, *prefix=None*, *file_name=None*, ***additional_kwargs*)

b2luigi.basf2_helper.targets module

class b2luigi.basf2_helper.targets.**ROOTLocalTarget**(*path=None*, *format=None*, *is_tmp=False*)

Bases: luigi.local_target.LocalTarget

exists()

Returns True if the path for this FileSystemTarget exists; False otherwise.

This method is implemented by using fs.

b2luigi.basf2_helper.tasks module

class b2luigi.basf2_helper.tasks.**Basf2FileMergeTask**(**args*, ***kwargs*)

Bases: *b2luigi.basf2_helper.tasks.MergerTask*

cmd = ['b2file-merge', '-f']

class b2luigi.basf2_helper.tasks.**Basf2PathTask**(**args*, ***kwargs*)

Bases: *b2luigi.basf2_helper.tasks.Basf2Task*

create_path()

max_event = <luigi.parameter.IntParameter object>

num_processes = <luigi.parameter.IntParameter object>

process()

Override this method with your normal run function. Do not touch the run function itself!

class b2luigi.basf2_helper.tasks.**Basf2Task**(**args*, ***kwargs*)

Bases: b2luigi.core.dispatchable_task.DispatchableTask

env

get_output_file_target(**args*, ***kwargs*)

get_serialized_parameters()

git_hash = <luigi.parameter.Parameter object>

class b2luigi.basf2_helper.tasks.**Basf2nTupleMergeTask**(**args*, ***kwargs*)

Bases: *b2luigi.basf2_helper.tasks.MergerTask*

cmd = ['fei_merge_files']

class b2luigi.basf2_helper.tasks.**HaddTask**(**args*, ***kwargs*)

Bases: *b2luigi.basf2_helper.tasks.MergerTask*

cmd = ['hadd', '-f']

class b2luigi.basf2_helper.tasks.**MergerTask**(**args*, ***kwargs*)

Bases: *b2luigi.basf2_helper.tasks.Basf2Task*

cmd = []

output()

The output that this Task produces.

The output of the Task determines if the Task needs to be run–the task is considered finished iff the outputs all exist. Subclasses should override this method to return a single Target or a list of Target instances.

Implementation note If running multiple workers, the output must be a resource that is accessible by all workers, such as a DFS or database. Otherwise, workers might compute the same output since they don't see the work done by other workers.

See Task.output

process()
Override this method with your normal run function. Do not touch the run function itself!

class b2luigi.basf2_helper.tasks.**SimplifiedOutputBasf2Task**(**args*, ***kwargs*)
Bases: *b2luigi.basf2_helper.tasks.Basf2PathTask*

create_path()

output()
The output that this Task produces.

The output of the Task determines if the Task needs to be run–the task is considered finished iff the outputs all exist. Subclasses should override this method to return a single `Target` or a list of `Target` instances.

Implementation note If running multiple workers, the output must be a resource that is accessible by all workers, such as a DFS or database. Otherwise, workers might compute the same output since they don't see the work done by other workers.

See Task.output

b2luigi.basf2_helper.utils module

b2luigi.basf2_helper.utils.**get_basf2_git_hash**()

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

4.7.1 Can I specify my own paths for the log files for tasks running on a batch system?

`b2luigi` will automatically create log files for the `stdout` and `stderr` output of a task processed on a batch system. The paths of these log files are defined relative to the location of the executed python file and contain the parameter of the task. In some cases one might one to specify other paths for the log files. To achieve this, a own `get_log_file_dir()` method of the task class must be implemented. This method must return a directory path for the stdout and the stderr files, for example:

```
class MyBatchTask(b2luigi.Task):
    ...
    def get_log_file_dir(self):
        filename = os.path.realpath(sys.argv[0])
        path = os.path.join(os.path.dirname(filename), "logs")
        return path
```

`b2luigi` will use this method if it is defined and write the log output in the respective files. Be careful, though, as these log files will of course be overwritten if more than one task receive the same paths to write to!

4.7.2 How do I handle parameter values which include "/" (or other unusual characters)?

`b2luigi` automatically generates the filenames for your output or log files out of the current tasks values in the form

```
<result-path>/param1=value1/param2=value2/..../filename.ext
```

The values are given by the serialisation of your parameter, which is basically its string representation. Sometimes, this representation may include characters not suitable for their usage as a path name, e.g. "/". Especially when you use a `DictParameter` or a `ListParameter`, you might not want to have its value in your output. Also, if you have credentials in the parameter (what you should never do of course!), you do not want to show them to everyone.

When using a parameter in *b2luigi* (or any of its derivatives), they have a new flag called `hashed` in their constructor, which makes the path creation only using a hashed version of your parameter value.

For example will this task:

```
class MyTask(b2luigi.Task):
    my_parameter = b2luigi.ListParameter(hashed=True)

    def run(self):
        with open(self.get_output_file_name("test.txt"), "w") as f:
            f.write("test")

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


if __name__ == "__main__":
    b2luigi.process(MyTask(my_parameter=["Some", "strange", "items", "with", "bad /␣
↪signs"]))
```

(continues on next page)

(continued from previous page)

create a file called `my_parameter=hashed_08928069d368e4a0f8ac02a0193e443b/test.txt` in your output folder instead of using the list value.

4.7.3 What does the ValueError "The task id {task.task_id} to be executed..." mean?

The *ValueError* exception *The task id <task_id> to be executed by this batch worker does not exist in the locally reproduced task graph.* is thrown by `b2luigi` batch workers if the task that should have been executed by this batch worker does not exist in the task graph reproduced by the batch worker. This means that the task graph produced by the initial `b2luigi.process` call and the one reproduced in the batch job differ from each other. This can be caused by a non-deterministic behavior of your dependency graph generation, such as a random task parameter.

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).

Python Module Index

b

Index

F

G

H

K

L

M

N

O

P

R

S

T

W