b2luigi Documentation

Release 0.7.0

Nils Braun

Jul 14, 2021

Contents

1 Why not use the already created batch tasks? 3

2 Is this the only thing I can do with b2luigi? 5

3 Why are you still talking, lets use it! 7

4 Content 9

4.1 Installation . . . 9
4.2 Quick Start . . . 10
4.3 Batch Processing . . . 14
4.4 Belle II specific examples . . . 23
4.5 API Documentation . . . 28
4.6 Run Modes . . . 38
4.7 FAQ . . . 40
4.8 Development and TODOs . . . 41

5 The name 43

6 The team 45

Python Module Index 47

Index 49

`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 (many many!) batch jobs in parallel** In other luigi batch 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, so 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

Is 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*.

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

Starting from release 04-00-00, *b2luigi* is already included in the externals. Follow this guid, if you want to update to the newest version nevertheless.

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 -U
```

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

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

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 *Belle II specific examples*.

4.2 Quick Start

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

Hint: The default batch system currently is LSF, so if you do not change it, LSF will be used. Check out *Batch Processing* for more information.

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:

```
import b2luigi
import random


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

    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.set_setting("result_dir", "results")
    b2luigi.process([MyNumberTask(some_parameter=i) for i in range(100)],
                    workers=200)
```

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.

Note: We have defined a result path in the script with

```
b2luigi.set_setting("results")
```

You can ignore that for not - we will come back to it later.

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).

Please not that we could have imported `b2luigi` with

```
import b2luigi as luigi
```

to make the transition between `b2luigi` and `luigi` even simpler.

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 `luigi.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 a single CPU power.

Hint: If you have no LSF queue ready or you do not want to run on the batch, you can also remove the *batch* argument. This will fall back to a normal `luigi` execution. Please see *Batch Processing* for more information on batch execution and the discussion of other batch systems.

---

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

   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 output file names 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. However this means, you always need to include all parameters in the file name to keep them different. This cumbersome work can be handled by `b2luigi` automatically , 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`.

```python
import b2luigi
import random


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

    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")


if __name__ == "__main__":
    b2luigi.set_setting("result_dir", "results")
    b2luigi.process([MyNumberTask(some_parameter=i) for i in range(100)],
                    workers=200)
```

Before continuing, remove the output of the former calculation.

```
rm -rf 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 and where the results are stored)
- The output for different parameters are stored on different locations, so no need to fear overriding results.
- The format of the folder structure makes it easy to work on it using bash commands as well as automated procedures.
- Other files related to your job, e.g. the submission files etc. are also placed into this folder (this is why the very first example defined it already).
- The default is to use the folder where your script is located.

Hint: In the example, the base path for the results is defined in the python file with

```
b2luigi.set_setting("result_dir", "results")
```

Instead, you can also add a `settings.json` with the following content in the folder where your script lives:

```
{
        "result_dir": "results"
}
```

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.

Attention: The result path (as well as any other paths, e.g. the log folders) are always evaluated relatively to your script file. This means `results` will always be created in the folder where your script is, not where your current working directory is. If you are unsure on the location, call

```
python3 simple-example.py --show-output
```

More on file systems is described in *Batch Processing*, which is also mostly true for non-batch calculations.

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:

```
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):
12         random_number = random.random()
13
14         with open(self.get_output_file_name("output_file.txt"), "w") as f:
15             f.write(f"{random_number}\n")
16
17
18 class MyAverageTask(b2luigi.Task):
19     def requires(self):
20         for i in range(100):
21             yield self.clone(MyNumberTask, some_parameter=i)
22
23     def output(self):
24         yield self.add_to_output("average.txt")
25
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.set_setting("result_dir", "results")
43     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 read some more documentation in *API Documentation* or have a look into the *FAQ*. Please also check out the different *Run Modes*.

4.3 Batch Processing

As shown in *Quick Start*, using the batch instead of local processing is really just a `--batch` on the command line or calling `process` with `batch=True`. However, there is more to discover!

4.3.1 Choosing the batch system

Using `b2luigi`'s settings mechanism (described here *`b2luigi.get_setting()`*) you can choose which batch system should be used. Currently, `htcondor` and `lsf` are supported. There is also an experimental wrapper for `gbasf2`, the Belle II submission tool for the LHC Worlwide Computing Grid, which works for `Basf2PathTask` tasks. More will come soon (PR welcome!).

4.3.2 Choosing the Environment

If you are doing a local calculation, all calculated tasks will use the same environment (e.g. `$PATH` setting, libraries etc.) as you have currently set up when calling your script(s). This makes it predictable and simple.

Things get a bit more complicated when using a batch farm, as the workers might not have the same environment set up, the batch submission does not copy the environment (or the local site administrators have forbidden that) or the system on the workers is so different that copying the environment from the scheduling machine does not make sense.

Therefore `b2luigi` provides you with three mechanism to set the environment for each task:

- You can give a bash script in the `env_script` setting (via `set_setting()`, `settings.json` or for each task as usual, see *`b2luigi.get_setting()`*), which will be called even before anything else on the worker. Use it to set up things like the path variables or the libraries (e.g. when you are using a virtual environment) and your batch system does not support environment copy from the scheduler to the workers. For example a useful script might look like this:

```
# Source my virtual environment
source venv/bin/activate
# Set some specific settings
export MY_IMPORTANT_SETTING 10
```

- You can set the `env` setting to a dictionary, which contains additional variables to be set up before your job runs. Using the mechanism described in *`b2luigi.get_setting()`* it is possible to make this task- or even parameter-dependent.
- By default, `b2luigi` re-uses the same `python` executable on the workers as you used to schedule the tasks (by calling your script). In some cases, this specific python executable is not present on the worker or is not usable (e.g. because of different operation systems or architectures). You can choose a new executable with the

`executable` setting (it is also possible to just use `python3` as the executable assuming it is in the path). The executable needs to be callable after your `env_script` or your specific `env` settings are used. Please note, that the `environment` setting is a list, so you need to pass your python executable with possible arguments like this:

```
b2luigi.set_setting("executable", ["python3"])
```

4.3.3 File System

Depending on your batch system, the filesystem on the worker processing the task and the scheduler machine can be different or even unrelated. Different batch systems and batch systems implementations treat this fact differently. In the following, the basic procedure and assumption is explained. Any deviation from this is described in the next section.

By default, `b2luigi` needs at least two folders to be accessible from the scheduling as well as worker machine: the result folder and the folder of your script(s). If possible, use absolute paths for the result and log directory to prevent any problems. Some batch systems (e.g. htcondor) support file copy mechanisms from the scheduler to the worker systems. Please checkout the specifics below.

Hint: All relative paths given to e.g. the `result_dir` or the `log_dir` are always evaluated relative to the folder where your script lives. To prevent any disambiguities, try to use absolute paths whenever possible.

Some batch system starts the job in an arbitrary folder on the workers instead of the current folder on the scheduler. That is why `b2luigi` will change the directory into the path of your called script before starting the job.

In case your script is accessible from a different location on the worker than on the scheduling machine, you can give the setting `working_dir` to specify where the job should run. Your script needs to be in this folder and every relative path (e.g. for results or log) will be evaluated from there.

4.3.4 Drawbacks of the batch mode

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

- You have to choose the queue/batch settings/etc. 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 just no way for `b2luigi` to know this beforehand.
- There is currently no 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 so far.

4.3.5 Batch System Specific Settings

Every batch system has special settings. You can look them up here:

LSF

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

Bases: *b2luigi.batch.processes.BatchProcess*

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

Additional to the basic batch setup (see *Batch Processing*), there are LSF-specific `settings`. These are:

- the LSF queue: `queue`.
- the LSF job name: `job_name`.

For example:

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

The default queue is the short queue `"s"`. If no `job_name` is set the task will appear as

```
<result_dir>/parameter1=value/.../executable_wrapper.sh"
```

when running `bjobs`.

- By default, the environment variables from the scheduler are copied to the workers. This also applies we start in the same working directory and can reuse the same executable etc. Normally, you do not need to supply `env_script` or alike.

HTCondor

class b2luigi.batch.processes.htcondor.**HTCondorProcess**(**args*, ***kwargs*)
Bases: *b2luigi.batch.processes.BatchProcess*

Reference implementation of the batch process for a HTCondor batch system.

Additional to the basic batch setup (see *Batch Processing*), additional HTCondor-specific things are:

- Please note that most of the HTCondor batch farms do not have the same environment setup on submission and worker machines, so you probably want to give an `env_script`, an `env setting` and/or a different `executable`.
- HTCondor supports copying files from submission to workers. This means if the folder of your script(s)/python project/etc. is not accessible on the worker, you can copy it from the submission machine by adding it to the setting `transfer_files`. This list can host both folders and files. Please note that due to HTCondors file transfer mechanism, all specified folders and files will be copied into the worker node flattened, so if you specify *a/b/c.txt* you will end up with a file *c.txt*. If you use the `transfer_files` mechanism, you need to set the `working_dir` setting to "." as the files will end up in the current worker scratch folder. All specified files/folders should be absolute paths.

  Hint: Please do not specify any parts or the full results folder. This will lead to unexpected behavior. We are working on a solution to also copy results, but until this the results folder is still expected to be shared.

  If you copy your python project using this setting to the worker machine, do not forget to actually set it up in your setup script. Additionally, you might want to copy your `settings.json` as well.
- Via the `htcondor_settings` setting you can provide a dict as a for additional options, such as requested memory etc. Its value has to be a dictionary containing HTCondor settings as key/value pairs. These options will be written into the job submission file. For an overview of possible settings refer to the HTCondor documentation.
- Same as for the LSF, the `job_name` setting allows giving a meaningful name to a group of jobs. If you want to be htcondor-specific, you can provide the `JobBatchName` as an entry in the

`htcondor_settings` dict, which will override the global `job_name` setting. This is useful for manually checking the status of specific jobs with

```
condor_q -batch <job name>
```

Example

```python
import b2luigi
import random


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

    htcondor_settings = {
        "request_cpus": 1,
        "request_memory": "100 MB"
    }

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

    def run(self):
        print("I am now starting a task")
        random_number = random.random()

        if self.some_parameter == 3:
            raise ValueError

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


class MyAverageTask(b2luigi.Task):
    htcondor_settings = {
        "request_cpus": 1,
        "request_memory": "200 MB"
    }

    def requires(self):
        for i in range(10):
            yield self.clone(MyNumberTask, some_parameter=i)

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

    def run(self):
        print("I am now starting the average task")

        # 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
```

(continues on next page)

(continued from previous page)

```
50
51          average = summed_numbers / counter
52
53          with open(self.get_output_file_name("average.txt"), "w") as f:
54              f.write(f"{average}\n")
55
56
57  if __name__ == "__main__":
58      b2luigi.process(MyAverageTask(), workers=200, batch=True)
```

GBasf2 Wrapper for LCG

class b2luigi.batch.processes.gbasf2.**Gbasf2Process**(**args*, ***kwargs*)

Bases: *b2luigi.batch.processes.BatchProcess*

Batch process for working with gbasf2 projects on the *LHC Computing Grid* (LCG).

Features

- **gbasf2 project submission**

  The gbasf2 batch process takes the basf2 path returned by the `create_path()` method of the task, saves it into a pickle file to the disk and creates a wrapper steering file that executes the saved path. Any basf2 variable aliases added in the `create_path()` method are also stored in the pickle file. It then sends both the pickle file and the steering file wrapper to the grid via the BelleII-specific Dirac-wrapper gbasf2.

- **Project status monitoring**

  After the project submission, the gbasf batch process regularly checks the status of all the jobs belonging to a gbasf2 project returns a success if all jobs had been successful, while a single failed job results in a failed project. You can close a running b2luigi process and then start your script again and if a task with the same project name is running, this b2luigi gbasf2 wrapper will recognize that and instead of resubmitting a new project, continue monitoring the running project.

  Hint: The outputs of gbasf2 tasks can be a bit overwhelming, so I recommend using the *central scheduler* which provides a nice overview of all tasks in the browser, including a status/progress indicator how many jobs in a gbasf2 project are already done.

- **Automatic download of datasets and logs**

  If all jobs had been successful, it automatically downloads the output dataset and the log files from the job sandboxes and automatically checks if the download was successful before moving the data to the final location. On failure, it only downloads the logs. The dataset download can be optionally disabled.

- **Automatic rescheduling of failed jobs**

  Whenever a job fails, gbasf2 reschedules it as long as the number of retries is below the value of the setting `gbasf2_max_retries`. It keeps track of the number of retries in a local file in the `log_file_dir`, so that it does not change if you close b2luigi and start it again. Of course it does not persist if you remove that file or move to a different machine.

Note: Despite all the automatization that this gbasf2 wrapper provides, the user is expected to have a basic understanding of how the grid works and know the basics of working with gbasf2 commands manually.

Caveats

- The gbasf2 batch process for luigi can only be used for tasks inhereting from `Basf2PathTask` or other tasks with a `create_path()` method that returns a basf2 path.
- It can be used **only for pickable basf2 paths**, with only some limited global basf2 state saved (currently aliases and global tags). The batch process stores the path created by `create_path` in a python pickle file and runs that on the grid. Therefore, **python basf2 modules are not yet supported**. To see if the path produced by a steering file is pickable, you can try to dump it with `basf2 --dump-path` and execute it again with `basf2 --execute-path`.
- Output format: Changing the batch to gbasf2 means you also have to adapt how you handle the output of your gbasf2 task in tasks depending on it, because the output will not be a single root file anymore (e.g. `B_ntuple.root`), but a collection of root files, one for each file in the input data set, in a directory with the base name of the root files, e.g.:

```
<task output directory>
            ├── B_ntuple.root
            │   └── B_ntuple_0.root
            │   └── B_ntuple_1.root
            │   └── ...
            ├── D_ntuple.root
            │   └── D_ntuple_0.root
            │   └── ...
```

Settings for gbasf2 tasks

To submit a task with the gbasf2 wrapper, you first you have to add the property `batch_system = "gbasf2"`, which sets the `batch_system` setting. It is not recommended to set that setting globally, as not all tasks can be submitted to the grid, but only tasks with a `create_path` method.

For gbasf2 tasks it is further required to set the settings

- `gbasf2_input_dataset`: String with the logical path of a dataset on the grid to use as an input to the task. You can provide multiple inputs by having multiple paths contained in this string, separated by commas without spaces. An alternative is to just instantiate multiple tasks with different input datasets, if you want to know in retrospect which input dataset had been used for the production of a specific output.
- `gbasf2_input_dslist`: Alternatively to `gbasf2_input_dataset`, you can use this setting to provide a text file containing the logical grid path names, one per line.
- `gbasf2_project_name_prefix`: A string with which your gbasf2 project names will start. To ensure the project associate with each unique task (i.e. for each of luigi parameters) is unique, the unique `task.task_id` is hashed and appended to the prefix to create the actual gbasf2 project name. Should be below 22 characters so that the project name with the hash can remain under 32 characters.

The following example shows a minimal class with all required options to run on the gbasf2/grid batch:

```
class MyTask(Basf2PathTask):
    batch_system = "gbasf2"
```

(continues on next page)

(continued from previous page)

```
    gbasf2_project_name_prefix = b2luigi.Parameter(significant=False)
    gbasf2_input_dataset = b2luigi.Parameter(hashed=True)
```

Other not required, but noteworthy settings are:

- `gbasf2_install_directory`: Defaults to `~/gbasf2KEK`. If you installed gbasf2 into another location, you have to change that setting accordingly.
- `gbasf2_release`: Defaults to the release of your currently set up basf2 release. Set this if you want the jobs to use another release on the grid.
- `gbasf2_print_status_updates`: Defaults to `True`. By setting it to `False` you can turn off the printing of of the job summaries, that is the number of jobs in different states in a gbasf2 project.
- `gbasf2_max_retries`: Default to 0. Maximum number of times that each job in the project can be automatically rescheduled until the project is declared as failed.
- `gbasf2_download_dataset`: Defaults to `True`. Disable this setting if you don't want to download the output dataset from the grid on job success. As you can't use the downloaded dataset as an output target for luigi, you should then use the provided `Gbasf2GridProjectTarget`, as shown in the following example:

```
from b2luigi.batch.processes.gbasf2 import get_unique_project_name,␣
↪Gbasf2GridProjectTarget

class MyTask(Basf2PathTask):
    # [...]
    def output(self):
        project_name = get_unique_project_name(self)
        return Gbasf2GridProjectTarget(project_name, task=self)
```

  This is useful when chaining gbasf2 tasks together, as they don't need the output locally but take the grid datasets as input. Also useful when you just want to produce data on the grid for other people to use.

- `gbasf2_download_logs`: Whether to automatically download the log output of gbasf2 projects when the task succeeds or fails. Having the logs is important for reproducibility, but k

The following optional settings correspond to the equally named `gbasf` command line options (without the `gbasf_` prefix) that you can set to customize your gbasf2 project:

`gbasf2_noscout`, `gbasf2_additional_files`, `gbasf2_input_datafiles`, `gbasf2_n_repition_job`, `gbasf2_force_submission`, `gbasf2_cputime`, `gbasf2_evtpersec`, `gbasf2_priority`, `gbasf2_jobtype`, `gbasf2_basf2opt`

It is further possible to append arbitrary command line arguments to the `gbasf2` submission command with the `gbasf2_additional_params` setting. If you want to blacklist a grid site, you can e.g. add

```
b2luigi.set_setting("gbasf2_additional_params",  "--banned_site LCG.KEK.jp")
```

Example Here is an example file to submit an analysis path created by the script in `examples/gbasf2/example_mdst_analysis` to grid via gbasf2:

Listing 1: File: `examples/gbasf2/gbasf2_example.py`

```python
import b2luigi
from b2luigi.basf2_helper.tasks import Basf2PathTask

import example_mdst_analysis


class MyAnalysisTask(Basf2PathTask):
    # set the batch_system property to use the gbasf2 wrapper batch process for this task
    batch_system = "gbasf2"
    # Must define a prefix for the gbasf2 project name to submit to the grid.
    # b2luigi will then add a hash derived from the luigi parameters to create a unique project name.
    gbasf2_project_name_prefix = b2luigi.Parameter()
    gbasf2_input_dataset = b2luigi.Parameter(hashed=True)
    # Example luigi cut parameter to facilitate starting multiple projects for different cut values
    mbc_lower_cut = b2luigi.IntParameter()

    def create_path(self):
        mbc_range = (self.mbc_lower_cut, 5.3)
        return example_mdst_analysis.create_analysis_path(
            d_ntuple_filename="D_ntuple.root",
            b_ntuple_filename="B_ntuple.root",
            mbc_range=mbc_range
        )

    def output(self):
        yield self.add_to_output("D_ntuple.root")
        yield self.add_to_output("B_ntuple.root")


class MasterTask(b2luigi.WrapperTask):
    """
    We use the MasterTask to be able to require multiple analyse tasks with
    different input datasets and cut values. For each parameter combination, a
    different gbasf2 project will be submitted.
    """

    def requires(self):
        input_dataset = \
            "/belle/MC/release-04-01-04/DB00000774/SkimM13ax1/prod00011778/e1003/4S/r00000/mixed/11180100/udst/sub00/"\
            "udst_000006_prod00011778_task10020000006.root"
        # if you want to iterate over different cuts, just add more values to this list
        mbc_lower_cuts = [5.15, 5.2]
        for mbc_lower_cut in mbc_lower_cuts:
            yield MyAnalysisTask(
                mbc_lower_cut=mbc_lower_cut,
                gbasf2_project_name_prefix="luigiExample",
                gbasf2_input_dataset=input_dataset,
                max_event=100,
            )

```

(continues on next page)

(continued from previous page)

```
51
52 if __name__ == '__main__':
53     main_task_instance = MasterTask()
54     n_gbasf2_tasks = len(list(main_task_instance.requires()))
55     b2luigi.process(main_task_instance, workers=n_gbasf2_tasks)
```

Handling failed jobs The gbasf2 input wrapper considers the gbasf2 project as failed if any of the jobs in the project failed and reached the maximum number of retries. It then automatically downloads the logs, so please look into them to see what the reason was. For example, it can be that only certain grid sites were affected, so you might want to exclude them by adding the `"--banned_site ...` to `gbasf2_additional_params`.

You also always reschedule jobs manually with the `gb2_job_reschedule` command or delete them with `gb2_job_delete` so that the gbasf2 batch process doesn't know they ever existed. Then run just run your luigi task/script again and it will start monitoring the running project again.

4.3.6 Add your own batch system

If you want to add a new batch system, all you need to do is to implement the abstract functions of `BatchProcess` for your system:

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 exactly 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 call a specific executable bash script only created for this task. It contains the setup of the environment (if given by the user via the settings), the change of the working directory (the directory of the python script or a specified directory by the user) and a call of this script with the current python interpreter (the one you used to call this main file or given by the setting `executable`) . However, we give this call 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:

```
/<path-to-your-exec>/python /your-project/some-file.py --batch-runner --task-
↪id MyTask_38dsf879w3
```

if the batch job should run the MyTask. The implementation of the abstract functions is responsible for creating an running the executable file and writing the log of the job into appropriate locations. You

can use the functions `create_executable_wrapper` and `get_log_file_dir` to get the needed information.

Checkout the implementation of the lsf task for some implementation example.

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 or idle. Will only be called after the job is started 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 (for b2luigi it makes no difference). 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 (maybe you need to check the exit code of your job).

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 also 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.

You can use the `b2luigi.core.utils.get_log_file_dir` and the `b2luigi.core.executable.create_executable_wrapper` functions to get the log base name and to create the executable script which you should call in your batch job.

After the start_job function is called by the framework (and no exception is thrown), it is assumed that a batch job is started or scheduled.

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_file_dir(self.task).

4.4 Belle II specific examples

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

4.4.1 Running at the NAF

The environment on the workers is different than on the scheduling machine, so we can not just copy the environment variables as on KEKCC.

You can use setup script (e.g. called `setup_basf2.sh`) with the following content

```
source /cvmfs/belle.cern.ch/tools/b2setup release-XX-XX-XX
```

All you have to do is specify the setup script as the `env_script` setting and also set the executable explicitly.

```
from time import sleep
import os

```

(continues on next page)

(continued from previous page)

```
import b2luigi


class MyTask(b2luigi.Task):
    parameter = b2luigi.IntParameter()

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

    def run(self):
        sleep(self.parameter)

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


class Wrapper(b2luigi.Task):
    def requires(self):
        for i in range(10):
            yield MyTask(parameter=i)

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

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


if __name__ == '__main__':
    # Choose htcondor as our batch system
    b2luigi.set_setting("batch_system", "htcondor")

    # Setup the correct environment on the workers
    b2luigi.set_setting("env_script", "setup_basf2.sh")

    # Most likely your executable from the submission node is not the same on
    # the worker node, so specify it explicitly
    b2luigi.set_setting("executable", ["python3"])

    # Where to store the results
    b2luigi.set_setting("result_dir", "results")

    b2luigi.process(Wrapper(), batch=True, workers=100)
```

Of course it is also possible to set those settings in the `settings.json` or as task-specific parameters. Please check out *b2luigi.get_setting()* for more information.

Please note that the called script as well as the results folder need to be accessible from both the scheduler and the worker machines. If needed, you can also include more setup steps in the source script.

4.4.2 Running at KEKCC

KEKCC uses LSF as the batch system. As this is the default for `b2luigi` there is nothing you need to do.

4.4.3 nTuple Generation

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

import basf2

import modularAnalysis


class AnalysisTask(Basf2PathTask):
    experiment_number = luigi.IntParameter()
    run_number = luigi.IntParameter()
    mode = luigi.Parameter()
    file_number = luigi.IntParameter()

    def output(self):
        # Define the outputs here
        yield self.add_to_output("D_n_tuple.root")
        yield self.add_to_output("B_n_tuple.root")

    def create_path(self):
        # somehow create filenames from parameters
        # self.experiment_number, self.run_number,
        # self.mode and self.file_number
        # (parameters just examples)
        input_file_names = ..

        path = basf2.create_path()
        modularAnalysis.inputMdstList('default', input_file_names, path=path)

        # Now fill your particle lists, just examples
        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)

        # When exporting, use the function get_output_file_name()
        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)
```

(continues on next page)

(continued from previous page)

```
        return path


class MasterTask(luigi.WrapperTask):
    def requires(self):
        # somehow loop over the runs, experiment etc.
            yield self.clone(AnalysisTask, experiment_number=...)


if __name__ == "__main__":
    luigi.process(MasterTask(), workers=500)
```

4.4.4 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 vertex
import generators
import reconstruction
from ROOT import Belle2


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:
            # in current main branch and release 5 the Y(4S)decay file is moved, so␣
↪try old and new locations
            find_file_ignore_error = True
            dec_file = Belle2.FileSystem.findFile('analysis/examples/tutorials/B2A101-
↪Y4SEventGeneration.dec',
                                                  find_file_ignore_error)
            if not dec_file:
                dec_file = Belle2.FileSystem.findFile('analysis/examples/simulations/
↪B2A101-Y4SEventGeneration.dec')
        elif self.event_type == SimulationType.continuum:
            dec_file = Belle2.FileSystem.findFile('analysis/examples/simulations/
↪B2A102-ccbarEventGeneration.dec')
```

(continues on next page)

(continued from previous page)

```
        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):
    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.matchMCTruth('D0', path=path)
        modularAnalysis.reconstructDecay('B- -> D0 pi-', '5.2 < Mbc < 5.3', path=path)
        try:  # treeFit is the new function name in light releases after release 4 (e.
↪g. light-2002-janus)
            vertex.treeFit('B+', 0.1, update_all_daughters=True, path=path)
        except AttributeError:  # vertexTree is the function name in release 4
            vertex.vertexTree('B+', 0.1, update_all_daughters=True, 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'],
```

(continues on next page)

(continued from previous page)

```
                                            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*, *ignore_additional_command_line_args=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()
```

(continues on next page)

(continued from previous page)

```
    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.
- **ignore_additional_command_line_args** (*bool, optional, default False*) – Ignore additional command line arguments. This is useful if you want to use this function in a file that also does some command line parsing.
- ****kwargs** – Additional keyword arguments passed to `luigi.build`.

Warning: You should always have just a single call to `process` in your script. If you need to have multiple calls, either use a *b2luigi.WrapperTask* or two scripts.

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*, see below) 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_dir/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_input_file_names_from_dict(*requirement_key*, *key=None*)

Get a dictionary of input file names of the tasks, which are defined in our requirements.

The requirement method should return a dict whose values are generator expressions (!) yielding required task objects.

Example

```
class TaskB(luigi.Task):

    def requires(self):
        return {
            "a": (TaskA(5.0, i) for i in range(100)),
            "b": (TaskA(1.0, i) for i in range(100)),
        }

    def run(self):
        result_a = do_something_with_a(
            self.get_input_file_names_from_dict("a")
        )
        result_b = do_something_with_b(
            self.get_input_file_names_from_dict("b")
        )

        combine_a_and_b(
            result_a,
            result_b,
            self.get_output_file_name("combined_results.txt")
        )

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

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

- **requirement_key** (str) – Specifies the required task expression.

- **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. It basically *emulates* a batch submission on your local computer (without any batch system) with the benefit of having a totally separete execution path. If your called task fails miserably (e.g. segfaults), it does not crash your main application.

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

Note: We are reusing the batch system implementation here, with all its settings and nobs to setup the environment etc. If you want to control it in more detail, please check out *Batch Processing*.

Implementation note: In the subprocess we are calling the current executable (which should by python) 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.

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*, *task=None*, *deprecated_keys=None*)
b2luigi adds a settings management to luigi and also uses it at various places. Many batch systems, the output and log path, the environment etc. is controlled via these settings.

There are four ways settings could be defined. They are used in the following order (an earlier setting overrides a later one):

1. If the currently processed (or scheduled) task has a property of the given name, it is used. Please note that you can either set the property directly, e.g.

```
class MyTask(b2luigi.Task):
    batch_system = "htcondor"
```

or by using a function (which might even depend on the parameters)

```
class MyTask(b2luigi.Task):
    @property
    def batch_system(self):
        return "htcondor"
```

The latter is especially useful for batch system specific settings such as requested wall time etc.

2. Settings set directly by the user in your script with a call to *b2luigi.set_setting()*.
3. Settings specified in the settings.json in the folder of your script *or any folder above that*. This makes it possible to have general project settings (e.g. the output path or the batch system) and a specific settings.json for your sub-project.

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.

Settings can be of any type, but are mostly strings.

Parameters

- **key** (str) – The name of the parameter to query.

- **task** – (*b2luigi.Task*): If given, check if the task has a parameter with this name.
- **default** (*optional*) – If there is no setting which the name, either return this default or if it is not set, raise a ValueError.
- **deprecated_keys** (List) – Former names of this setting, will throw a warning when still used

b2luigi.**set_setting**(*key*, *value*)
Set the setting with the specified name - overriding any setting.json. If you want to have task specific settings, create a parameter with the given name or your task.

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 so called thanksgiving bug (see https://www.arashrouhani.com/luigi-budapest-bi-oct-2015/#/21). It occurs, when you define an output of a task and in its run function, you create this output before filling it with content (maybe even only after a long lasting calculation). It may happen, that during the creation of the output and the finish of the calculation some other tasks checks if the output is already there, finds it and assumes, that the task is already finished (although there is probably only non-sense in the file so far).

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, which 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.**product_dict**(***kwargs*)
Cross-product the given parameters and return a list of dictionaries.

Example

```
>>> 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:

```
def requires(self):
    for args in product_dict(arg_1=[1, 2], arg_2=[3, 4]):
        yield some_task(**args)
```

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.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`

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
Command to use to merge basf2 tuple 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` (see *Batch Processing*) 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, and call the `dry_run` method of the task if implemented:

  ```
  class SomeTask(b2luigi.Task):
      [...]
      def dry_run(self):
          # if a method with this name is provided, it will be executed
          # automatically when starting the processing in dry-run mode
          do_some_stuff_in_dry_run_mode()
  ```

  This feature can be easily used for e.g. file name debugging, i.e. to print out the file names `b2luigi` will create when running the actual task. The exit code of the `dry-run` mode 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.

4.6.1 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` (the parent project of `b2luigi`) brings a nice visualisation and scheduling 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.

4.7 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 Can I exclude one job from batch processing

The setting `batch_system` defines which submission method is used for scheduling your tasks when using `batch=True` or `--batch`. In most cases, you set your `batch_system` globally (e.g. in a `settings.json`) file and start all your tasks with `--batch` or `batch=True`. If you want a single task to run only locally (e.g. because of constraints in the batch farm) you can set the `batch_system` only for this job by adding a member to this task:

```
class MyLocalTask(b2luigi.Task):
    batch_system = "local"

    def run(self):
        ...
```

4.7.3 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"]))
```

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

4.7.4 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
```

Now you can start hacking and your changes will be immediately available to you.

4. Install pre-commit, which automatically checks your code

```
pip3 [ --user ] install pre-commit
pre-commit install  # install the pre-commit hooks
pre-commit  # run pre-commit manually, checks all staged ("added") changes
```

In particular, the python files are checked with flake8 for syntax and PEP 8 style errors. I recommend using an IDE or editor which automatically highlights errors with flake8 or a similar python linter (e.g. pylint).

5. We use the unittest package for testing some parts of the code. All tests reside in the `tests/` sub-directory. To run all tests, run the command

```
python3 -m unittest
```

in the root of `b2luigi` repository. If you add some functionality, try to add some tests for it.

6. The documentation is hosted on readthedocs and build automatically on every commit to main. 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.

6. If you are a core developer and want to release a new version:

   a. Make sure all changes are committed and merged on main

   b. Use the bump2version 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. Create a new release with a description on github

   e. Check that the new release had been published to PyPi, which should happen automatically via github actions. Alternatively, you can also manually publish a release via

```
flit publish
```

4.8.2 Open TODOs

For a list of potential features, improvements and bugfixes see the github issues. Help is welcome, so feel free to pick one, e.g. with the `good first issue` or `help wanted` tags.

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 Michael Eliachevitch (meliache)

Original author Nils Braun (nils-braun)

Features, fixing, help and testing

- Felix Metzner (FelixMetzner)
- Patrick Ecker (eckerpatrick)
- Jochen Gemmler
- Maximilian Welsch (welschma)
- Kilian Lieret (klieret)
- Sviatoslav Bilokin (bilokin)
- Phil Grace (philiptgrace)
- Anselm Baur (anselmbaur)

Stolen ideas

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

Python Module Index

b

b2luigi.basf2_helper.data, 35
b2luigi.basf2_helper.targets, 37
b2luigi.basf2_helper.tasks, 37
b2luigi.basf2_helper.utils, 38

Index

A

add_to_output() (*b2luigi.Task method*), 30

B

b2luigi.basf2_helper.data (*module*), 35
b2luigi.basf2_helper.targets (*module*), 37
b2luigi.basf2_helper.tasks (*module*), 37
b2luigi.basf2_helper.utils (*module*), 38
Basf2FileMergeTask (*class in b2luigi.basf2_helper.tasks*), 37
Basf2nTupleMergeTask (*class in b2luigi.basf2_helper.tasks*), 38
Basf2PathTask (*class in b2luigi.basf2_helper.tasks*), 37
Basf2Task (*class in b2luigi.basf2_helper.tasks*), 37
BatchProcess (*class in b2luigi.batch.processes*), 22

C

cdst (*b2luigi.basf2_helper.data.DataMode attribute*), 35
CdstDataTask (*class in b2luigi.basf2_helper.data*), 35
clear_setting() (*in module b2luigi*), 34
clone_on_cdst() (*in module b2luigi.basf2_helper.data*), 37
clone_on_mdst() (*in module b2luigi.basf2_helper.data*), 37
clone_on_raw() (*in module b2luigi.basf2_helper.data*), 37
clone_on_skimmed_raw() (*in module b2luigi.basf2_helper.data*), 37
cmd (*b2luigi.basf2_helper.tasks.Basf2FileMergeTask attribute*), 37
cmd (*b2luigi.basf2_helper.tasks.Basf2nTupleMergeTask attribute*), 38
cmd (*b2luigi.basf2_helper.tasks.HaddTask attribute*), 38
cmd (*b2luigi.basf2_helper.tasks.MergerTask attribute*), 38
create_path() (*b2luigi.basf2_helper.tasks.Basf2PathTask method*), 37
create_path() (*b2luigi.basf2_helper.tasks.SimplifiedOutputBasf2Task method*), 38

D

data_mode (*b2luigi.basf2_helper.data.CdstDataTask attribute*), 35
data_mode (*b2luigi.basf2_helper.data.DataTask attribute*), 35
data_mode (*b2luigi.basf2_helper.data.MdstDataTask attribute*), 36
data_mode (*b2luigi.basf2_helper.data.RawDataTask attribute*), 36
data_mode (*b2luigi.basf2_helper.data.SkimmedRawDataTask attribute*), 36
database (*b2luigi.basf2_helper.data.DstDataTask attribute*), 36
DataMode (*class in b2luigi.basf2_helper.data*), 35
DataTask (*class in b2luigi.basf2_helper.data*), 35
dispatch() (*in module b2luigi*), 32
DispatchableTask (*class in b2luigi*), 32
DstDataTask (*class in b2luigi.basf2_helper.data*), 36

E

exists() (*b2luigi.basf2_helper.targets.ROOTLocalTarget method*), 37
experiment_number (*b2luigi.basf2_helper.data.DataTask attribute*), 35
ExternalTask (*class in b2luigi*), 32

F

file_name (*b2luigi.basf2_helper.data.DataTask attribute*), 35

G

Gbasf2Process (*class in b2luigi.batch.processes.gbasf2*), 18
get_basf2_git_hash() (*in module b2luigi.basf2_helper.utils*), 38

get_input_file_names() (*b2luigi.Task method*), 31
get_input_file_names_from_dict() (*b2luigi.Task method*), 31
get_job_status() (*b2luigi.batch.processes.BatchProcess method*), 23
get_output_file_name() (*b2luigi.Task method*), 32
get_output_file_target() (*b2luigi.basf2_helper.tasks.Basf2Task method*), 37
get_serialized_parameters() (*b2luigi.basf2_helper.tasks.Basf2Task method*), 37
get_setting() (*in module b2luigi*), 33
git_hash (*b2luigi.basf2_helper.tasks.Basf2Task attribute*), 38

H

HaddTask (*class in b2luigi.basf2_helper.tasks*), 38
HTCondorProcess (*class in b2luigi.batch.processes.htcondor*), 16

K

kill_job() (*b2luigi.batch.processes.BatchProcess method*), 23

L

LSFProcess (*class in b2luigi.batch.processes.lsf*), 15

M

max_event (*b2luigi.basf2_helper.tasks.Basf2PathTask attribute*), 37
mdst (*b2luigi.basf2_helper.data.DataMode attribute*), 35
MdstDataTask (*class in b2luigi.basf2_helper.data*), 36
MergerTask (*class in b2luigi.basf2_helper.tasks*), 38

N

num_processes (*b2luigi.basf2_helper.tasks.Basf2PathTask attribute*), 37

O

on_temporary_files() (*in module b2luigi*), 34
output() (*b2luigi.basf2_helper.data.CdstDataTask method*), 35
output() (*b2luigi.basf2_helper.data.DstDataTask method*), 36
output() (*b2luigi.basf2_helper.data.MdstDataTask method*), 36
output() (*b2luigi.basf2_helper.data.RawDataTask method*), 36
output() (*b2luigi.basf2_helper.data.SkimmedRawDataTask method*), 36
output() (*b2luigi.basf2_helper.tasks.MergerTask method*), 38
output() (*b2luigi.basf2_helper.tasks.SimplifiedOutputBasf2Task method*), 38

P

prefix (*b2luigi.basf2_helper.data.DataTask attribute*), 35
process() (*b2luigi.basf2_helper.tasks.Basf2PathTask method*), 37
process() (*b2luigi.basf2_helper.tasks.MergerTask method*), 38
process() (*b2luigi.DispatchableTask method*), 33
process() (*in module b2luigi*), 28
prod (*b2luigi.basf2_helper.data.DstDataTask attribute*), 36
product_dict() (*in module b2luigi.core.utils*), 34

R

raw (*b2luigi.basf2_helper.data.DataMode attribute*), 35
RawDataTask (*class in b2luigi.basf2_helper.data*), 36
release (*b2luigi.basf2_helper.data.DstDataTask attribute*), 36
ROOTLocalTarget (*class in b2luigi.basf2_helper.targets*), 37
run_number (*b2luigi.basf2_helper.data.DataTask attribute*), 35

S

set_setting() (*in module b2luigi*), 34
SimplifiedOutputBasf2Task (*class in b2luigi.basf2_helper.tasks*), 38
skimmed_raw (*b2luigi.basf2_helper.data.DataMode attribute*), 35
SkimmedRawDataTask (*class in b2luigi.basf2_helper.data*), 36
start_job() (*b2luigi.batch.processes.BatchProcess method*), 23

T

Task (*class in b2luigi*), 30

W

WrapperTask (*class in b2luigi*), 32