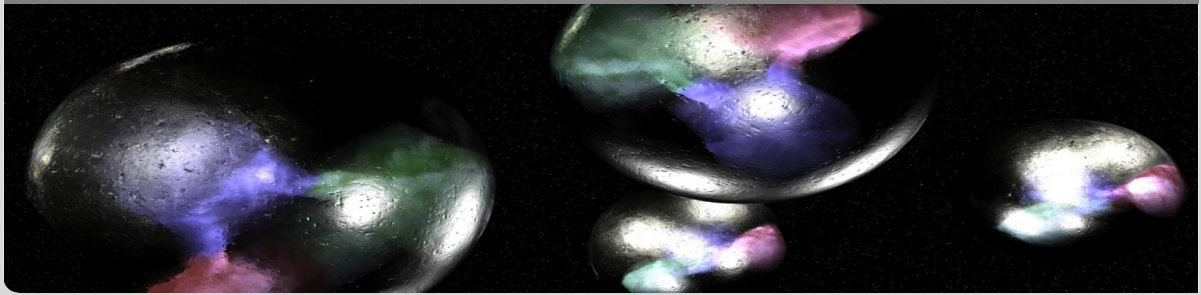
Luigi and B2Luigi

Belle II Software and Performance Workshop

Felix Metzner | 10th March 2020

INSTITUT FÜR EXPERIMENTELLE TEILCHENPHYSIK (ETP) FELIX.METZNER@KIT.EDU

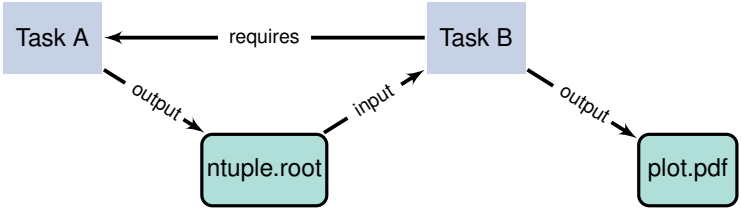


What is Luigi?

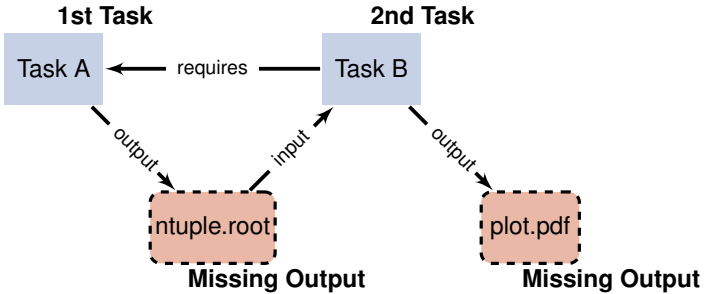
Luigi is a framework for the management of complex workflows provided as Python package.

It allows to define elaborate data processing flows by

- splitting the workflow into separate **tasks**
- which can depend on the **output** of other tasks
- and **parameters**.



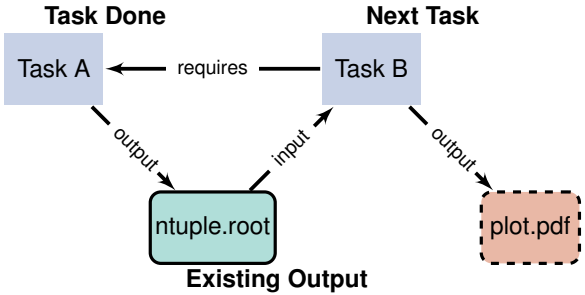
What is Luigi?



Luigi is aware of

- **dependencies among tasks**,
- failed tasks (output does not exist),
- already fulfilled tasks when restarting the program and
- changed dependencies (to other tasks or parameters).

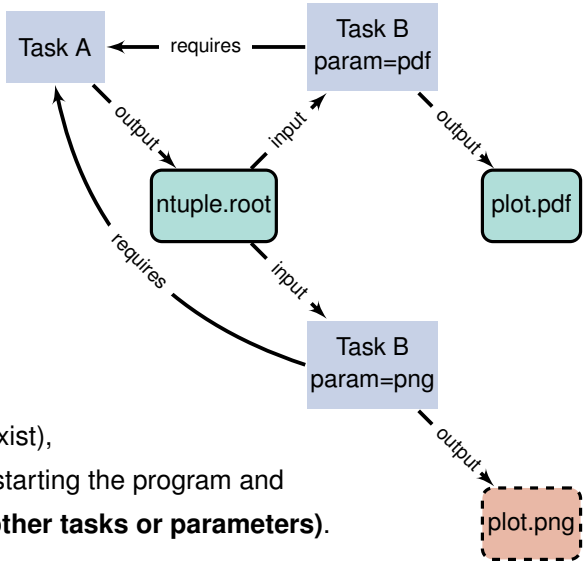
What is Luigi?



Luigi is aware of

- dependencies among tasks,
- **failed tasks (output does not exist)**,
- **already fulfilled tasks when restarting the program** and
- changed dependencies (to other tasks or parameters).

What is Luigi?



Luigi is aware of

- dependencies among tasks,
- failed tasks (output does not exist),
- already fulfilled tasks when restarting the program and
- **changed dependencies (to other tasks or parameters)**.

A Luigi Task

```
import luigi

class MyTask(luigi.Task):
    param = luigi.IntParameter(default=42)

    def requires(self):
        return SomeOtherTask(self.param)

    def run(self):
        with self.output().open("w") as f:
            f.write("hello world")

    def output(self):
        return luigi.LocalTarget(f"/temp/foo/output_{self.param}.txt")

if __name__ == "__main__":
    luigi.run()
```

Parameters of this Task

What other Tasks it depends on

The Task's business logic

The output of this Task

Running Luigi

There are various ways to invoke a Luigi process:

In the `__main__` method:

```
luigi.run()
```

In the `__main__` method and starting a specific Task:

```
luigi.process(MyTask(param=47), workers=1)
```

From the command line:

```
luigi --module my_tasks MyTask --local-scheduler --param=1337
```

Luigi Parameters

Parameters are attributes of a Task class which can be used to parameterize a Task.

The user must define how the parameters affect the business logic and the output of the Task.

```
class TaskB(luigi.Task):
    param = luigi.Parameter(default="pdf")
    ...
    def run(self):
        Some code creating a plot...
        if self.param == "pdf":
            fig.savefig("path/to/plot.pdf")
        else:
            fig.savefig("path/to/plot.png")
```

Depending on the data type the appropriate parameter class must be used:

- `luigi.Parameter` for strings
- `luigi.IntParameter` for integers
- `luigi.FloatParameter` for floats
- `luigi.BoolParameter` for booleans
- `luigi.DictParameter` for dictionaries
- and many more...

Dependencies

If a Task B requires another Task A, this must be defined in the `requires` method of Task B:

```
class TaskB(luigi.Task):
    ...
    def requires(self):
        yield TaskA()
```

This means Task B requires the output of Task A.
⇒ Task B depends on Task A.
⇒ Task B uses the output of Task A.
⇒ Task B is executed after Task A.

If the dependency is **simple** this can be achieved with a **decorator**.
Simple means: Task A and Task B have the exact same parameters.

```
from luigi.util import requires

@requires(TaskA)
class TaskB(luigi.Task):
    ...
    # No requires method necessary!
```

Requiring Multiple Tasks and the WrapperTask

Requiring Multiple Tasks

```
class CalibrationEvaluationTask(luigi.Task):
    ...
    def requires(self):
        for in_file in mdst_files:
            yield ReconstructionTask(mdst_file=in_file)
        yield CalibrationTask(calibration_tag="cal_XYZ")
```

The `WrapperTask` Class

```
class CalibrationMasterTask(luigi.WrapperTask):
    def requires(self):
        calibration_tags = ["cal_ABC", "cal_XYZ"]
        for calibration_tag in calibration_tags:
            yield CalibrationEvaluationTask(calibration_tag)
        yield CalibrationComparisonTask(calibration_tags)
```

Parameter Passing

When invoking the Luigi process

This option is only valid for the **last Task** of a chain. E.g.:

- in the `__main__` method and starting a specific Task:

```
luigi.process(MyTask(param=47), workers=1)
```

- or from the command line:

```
luigi --module my_tasks MyTask --local-scheduler --param=1337
```

In the `requires` method

Via decorators and cloning

Parameter Passing

When invoking the Luigi process

In the `requires` method

Usually necessary when

- the required Task has more parameters then the requiring one, or
- a Task is required multiple time with different parameters.

```
class CalibrationMasterTask(luigi.WrapperTask):
    def requires(self):
        calibration_tags = ["cal_ABC", "cal_XYZ"]
        for calibration_tag in calibration_tags:
            yield CalibrationEvaluationTask(calibration_tag)
```

Via decorators and cloning

Parameter Passing

When invoking the Luigi process

In the `requires` method

Via decorators and cloning

If the Tasks have all parameters in common and ⇒ use the `@requires` decorator.

If only some parameters are shared ⇒ use the `clone()` method of the `luigi.Task` class:

```
class TaskB(luigi.Task):
    def requires(self):
        yield self.clone(TaskA, param_A=42)
```

Bonus: Use the `@inherits` decorator to avoid the need to define the same parameter multiple times for each Task!

Parameter Passing

When invoking the Luigi process

In the `requires` method

Via decorators and cloning

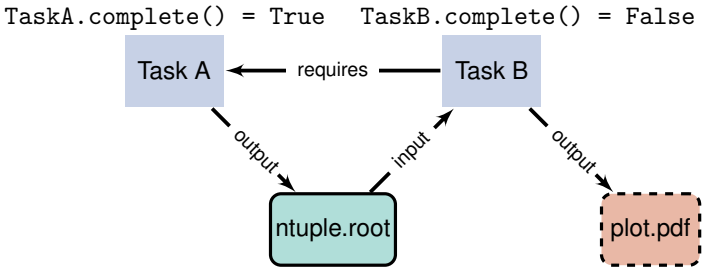
See also this link to explore the many other well documented options. . .

Luigi Targets

“A Target is a resource generated by a Task”

- A Task can create one or multiple Targets as output.
- If all Target defined in the output method of a Task exist, the Task is completed.
- A output Target of Task A can be the input to a Task B which requires Task A.

The most basic implementation is the `luigi.LocalTarget` class.



Luigi Targets

You normally will have to deal with Targets in the following methods of your Task:

Output Method — The providing side

Defines the output files a task will generate via

- list of `luigi.Target` objects
- generator of `luigi.Target` objects
- dictionary mapping keys to `luigi.Target` objects
- ...

```python
def output(self):
    return luigi.LocalTarget(f"/temp/output.txt")
```

```python
def output(self):
    yield luigi.LocalTarget(f"/temp/output1.txt")
    yield luigi.LocalTarget(f"/temp/output2.txt")
```

```python
def output(self):
    key = f"output1")
    file = luigi.LocalTarget(f"/temp/output1.txt")
    return {key:  file}
```

Input Method — The receiving side

Luigi Targets

You normally will have to deal with Targets in the following methods of your Task:

Output Method — The providing side

Input Method — The receiving side

A Task's `self.input()` method provides access to the Targets output by all required Tasks.

The object returned by `self.input()` depends on:

- the `requires` method of the Task and
- the `output` method of the required Tasks.

For a LocalTarget: Remember to use the `path` property to access the actual path of the target!

Example: A Task requires multiple other Tasks, each of which returns a dictionary of keys and `luigi.Target` objects. To convert this to a dictionary containing the actual paths, use

```
path_dict = {k: v.path for d in self.input() for k, v in d.items()}
```

Atomic Output

Writing larger files (e.g. `ROOT`-file) to disk takes time.
If this process is interrupted unexpectedly, the **partly written file still exists**.
$\Rightarrow$ Luigi might assume the Task was **completed successfully**, because the output exists!

To avoid this, make the creation of the final file atomic!

Some options to achieve this are:

- write to a temporary file first, then **rename or link** this file to the expected output path;
- write to your target file in the scope of `Luigi.LocalTarget.open(mode="w")`:

```
with self.output().open("w") as f:
    f.write("hello world")
```

Some General Advise

Use Luigi only to define the dependencies and the data processing flow!

Define your business logic in dedicated classes or functions independent of Luigi. These are then used in the `run` method of a Task, but can also be used separately

This ensures

- that you can use the code also elsewhere;
- a better maintainability of the code;
- that the code can be tested easier;
- a cleaner code...

Luigi Workers and the Luigi Scheduler

Once the workflow is defined, the Task chain can be executed...

...but what is the best way to do this?

Luigi Workers — Running with multiple **local** processes

Luigi Scheduler — Keeping track of your Tasks

Luigi Workers and the Luigi Scheduler

Luigi Workers — Running with multiple **local** processes

Processing multiple independent Tasks in parallel is easy.
Just start Luigi with **multiple workers**, e.g.:

```
luigi.process(MyTask(param=47), workers=10)
```

Luigi Scheduler — Keeping track of your Tasks

Luigi Workers and the Luigi Scheduler

Luigi Workers — Running with multiple **local** processes

Luigi Scheduler — Keeping track of your Tasks

When you run Luigi on **multiple machines**, e.g. via cron jobs, **Luigi's Scheduler** is necessary to manage the work distribution:

Start the Scheduler server (e.g. on KEKcc's `cw01`) with

```
luigid --background --port <my_port>
```

and your worker(s) with

```
python3 your_script.py --scheduler-host cw01 --scheduler-port <my_port>
```

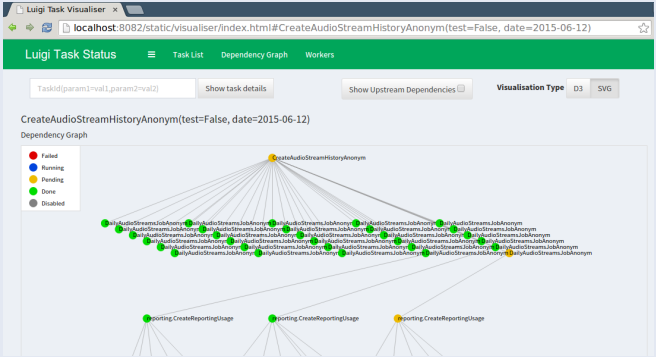
Luigi Workers and the Luigi Scheduler

Luigi Workers — Running with multiple **local** processes

Luigi Scheduler — Keeping track of your Tasks

Opening the **Luigi Task Visualiser** via `localhost:<my_port>` in a browser will give you a nice overview!

Don't forget to forward the port. . . !



Luigi Workers and the Luigi Scheduler

Once the workflow is defined, the Task chain can be executed...

...but what is the best way to do this?

Luigi Workers — Running with multiple **local** processes

Luigi Scheduler — Keeping track of your Tasks

Luigi also provides support for many batch systems, but not the ones commonly used in physics...

Enter B2Luigi

B2Luigi is an extension of the Luigi project developed by **Nils Braun**[1].

It provides additional features which can be categorized into three groups:

basf2 Features

B2Luigi introduces basf2 related Task classes such as the

- **Basf2PathTask** which will run a basf2 path, or the
- **Basf2nTupleMergeTask**, a wrapper task to merge `ROOT`-files produced by other Tasks.

Batch System Features

It adds back ends to the batch systems we usually use: **LSF @ KEKcc** or **HTCondor @ NAF**

Bread and Butter Features

B2Luigi can take care of basic bookkeeping for you, e.g. handling the output and input.

[1] nilslennartbraun@gmail.com

Bread and Butter Features

First and foremost, B2Luigi provides a **drop in replacement** for the basic luigi.Task class.

This and other **Super Hero Task classes** can be used directly

```
import b2luigi
class MyTask(b2luigi.Task)
```

or replace the respective Luigi classes by importing B2Luigi via

```
import b2luigi as luigi
```

Bread and Butter Features

The `b2luigi.Task` class will take care of the output- and input paths for you via the methods:

- `self.add_to_output(output_file_name="output.txt")`
- `self.get_input_file_names(key="previous_output.txt")`
- `self.get_output_file_name(key="output.txt")`

You only have to specify the file name itself.
The directory structure will be

- created automatically **at the location of the executed script** and
- dictated by the **git-hash of the used basf2 version**, as well as
- the **parameters** of the Tasks.

Bread and Butter Features

B2Luigi also adds handy **command line options**, such as
`--show-output` to get an overview of the **output that will be produced**:

```
python3 basf2_chain_example.py --show-output
D_n_tuple.root
  /path/to/script/location/git_hash=<hash>/n_events=1/D_n_tuple.root
  /path/to/script/location/git_hash=<hash>/n_events=1/event_type=y4s/D_n_tuple.root
  /path/to/script/location/git_hash=<hash>/n_events=1/event_type=continuum/D_n_tuple.root
B_n_tuple.root
  /path/to/script/location/git_hash=<hash>/n_events=1/B_n_tuple.root
  /path/to/script/location/git_hash=<hash>/n_events=1/event_type=y4s/B_n_tuple.root
  /path/to/script/location/git_hash=<hash>/n_events=1/event_type=continuum/B_n_tuple.root
reconstructed_output.root
  /path/to/script/location/git_hash=<hash>/n_events=1/event_type=y4s/reconstructed_output.root
  /path/to/script/location/git_hash=<hash>/n_events=1/event_type=continuum/reconstructed_output.root
simulation_full_output.root
  /path/to/script/location/git_hash=<hash>/n_events=1/event_type=y4s/simulation_full_output.root
  /path/to/script/location/git_hash=<hash>/n_events=1/event_type=continuum/simulation_full_output.root
```

You can test them easily yourself on the scripts provided in the **B2Luigi examples**!

Note that you have to use B2Luigi's `b2luigi.process` method to enable these options!

Bread and Butter Features

B2Luigi also adds handy **command line options**, such as
`--dry-run` (similar to luigi option) to get an overview of the **Tasks that will be executed**:

```
python3 basf2_chain_example.py --dry-run
AnalysisTask
  Would run AnalysisTask(git_hash=<hash>, n_events=1, event_type=y4s)
  Would run AnalysisTask(git_hash=<hash>, n_events=1, event_type=continuum)

MasterTask
  Would run MasterTask(git_hash=<hash>, n_events=1)

ReconstructionTask
  Would run ReconstructionTask(git_hash=<hash>, n_events=1, event_type=y4s)
  Would run ReconstructionTask(git_hash=<hash>, n_events=1, event_type=continuum)

SimulationTask
  Would run SimulationTask(git_hash=<hash>, n_events=1, event_type=y4s)
  Would run SimulationTask(git_hash=<hash>, n_events=1, event_type=continuum)
```

You can test them easily yourself on the scripts provided in the **B2Luigi examples**!

Note that you have to use B2Luigi's `b2luigi.process` method to enable these options!

Batch Systems

LSF (e.g. at KEKcc)

This is B2Luigi's default setting for running on a batch system!
All you need to do is to start the process with the **batch** run mode

- either via the command line option `--batch`:

```
python3 my_b2luigi_script.py --batch
```

- or by specifying the run mode batch in the `process` call in the `__main__` function:

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

The **LSF queue** to be used can be specified via the Task parameter `queue` (the default is `"s"`):

```
class MyLongTask(b2luigi.Task):
    queue = "l"  # Task will be submitted to long queue in batch mode!
```

HTCondor (e.g. at NAF)

Note that you have to use B2Luigi's `b2luigi.process` method to enable these options!

Batch Systems

LSF (e.g. at KEKcc)

HTCondor (e.g. at NAF)

Using HTCondor requires some settings to be defined (see also B2Luigi documentation):

The settings required for HTCondor can be set

- directly in the script using `b2luigi.set_setting` (see example on the right), or
- in the B2Luigi `setting.json` file in the folder of the script.

Task specific settings can be defined directly in the Task class, as described here

```
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")

    # Specifying executable for worker node explicitly
    b2luigi.set_setting("executable", ["python3"])

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

    b2luigi.process(MasterTask(), batch=True, workers=10)
```

Note that you have to use B2Luigi's `b2luigi.process` method to enable these options!

A basf2 Example

To explaine some of the basf2 specific Task classes, such as

- the **Basf2PathTask** and
- the **Basf2nTupleMergeTask**

we will go through a **basf2 specific B2Luigi example** describing a B2Luigi workflow for

- Monte Carlo simulation,
- reconstruction,
- a simple analysis
- and the merging of the resulting n-tuples.

You can find the source file of the example here.

A basf2 Example

Given the described steps we have the following basic structure of Tasks:

- Monte Carlo simulation,
- reconstruction,
- a simple analysis
- and the merging of the resulting n-tuples,

```
class SimulationTask(b2luigi.Basf2PathTask): # First task, requires nothing
    ...

@luigi.requires(SimulationTask) # Requires the output of SimulationTask
class ReconstructionTask(b2luigi.Basf2PathTask):
    ...

@luigi.requires(ReconstructionTask) # Requires the output of ReconstructionTask
class AnalysisTask(b2luigi.Basf2PathTask):
    ...

# Entry Task; requires all AnalysisTasks and merges their output
class MasterTask(b2luigi.Basf2nTupleMergeTask):
    ...
```

A basf2 Example

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

class SimulationTask(Basf2PathTask): # First task, requires nothing
  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:
      df = Belle2.FileSystem.findFile("analysis/examples/tutorials/B2A101-Y4SEventGeneration.dec")
    elif self.event_type == SimulationType.continuum:
      df = Belle2.FileSystem.findFile("analysis/examples/simulations/B2A102-ccbarEventGeneration.dec")
    else:
      raise ValueError(f"Event type {self.event_type} is not valid.")

    generators.add_evtgen_generator(path, "signal", signaldecfile=df)
    modularAnalysis.loadGearbox(path)
    simulation.addsimulation(path)
    path.add_module("RootOutput", outputFileName=self.get_output_file_name("simulation_output.root"))
    return path

  def output(self):
    yield self.add_to_output("simulation_output.root")
```

We use B2Luigi's `Basf2PathTask` class

A basf2 Example

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

class SimulationTask(Basf2PathTask): # First task, requires nothing
  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:
      df = Belle2.FileSystem.findFile("analysis/examples/tutorials/B2A101-Y4SEventGeneration.dec")
    elif self.event_type == SimulationType.continuum:
      df = Belle2.FileSystem.findFile("analysis/examples/simulations/B2A102-ccbarEventGeneration.dec")
    else:
      raise ValueError(f"Event type {self.event_type} is not valid.")

    generators.add_evtgen_generator(path, "signal", signaldecfile=df)
    modularAnalysis.loadGearbox(path)
    simulation.addsimulation(path)
    path.add_module("RootOutput", outputFileName=self.get_output_file_name("simulation_output.root"))
    return path

  def output(self):
    yield self.add_to_output("simulation_output.root")
```

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

A basf2 Example

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

class SimulationTask(Basf2PathTask): # First task, requires nothing
  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:
      df = Belle2.FileSystem.findFile("analysis/examples/tutorials/B2A101-Y4SEventGeneration.dec")
    elif self.event_type == SimulationType.continuum:
      df = Belle2.FileSystem.findFile("analysis/examples/simulations/B2A102-ccbarEventGeneration.dec")
    else:
      raise ValueError(f"Event type {self.event_type} is not valid.")

    generators.add_evtgen_generator(path, "signal", signaldecfile=df)
    modularAnalysis.loadGearbox(path)
    simulation.addsimulation(path)
    path.add_module("RootOutput", outputFileName=self.get_output_file_name("simulation_output.root"))
    return path

  def output(self):
    yield self.add_to_output("simulation_output.root")
```

The `create_path()` method replaces Luigi's `run()`
It returns the basf2 path that will be processed.

A basf2 Example

```
@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_output.root"))
    modularAnalysis.loadGearbox(path)
    reconstruction.add_reconstruction(path)

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

    return path

  def output(self):
    yield self.add_to_output("reco_output.root")
```

A basf2 Example

```
@luigi.requires(ReconstructionTask)
class AnalysisTask(Basf2PathTask):
  def create_path(self):
    path = basf2.create_path()
    modularAnalysis.inputMdstList("default", self.get_input_file_names("reco_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)
    D_file = self.get_output_file_name("D_ntuple.root")
    D_vars = ["M", "p", "E", "useCMSFrame(p)", "useCMSFrame(E)", "daughter(0, kaonID)", ...]
    modularAnalysis.variablesToNtuple("D0", D_vars, filename=D_file, path=path)
    B_file = self.get_output_file_name("B_ntuple.root")
    B_vars = ["Mbc", "deltaE", "isSignal", "mcErrors", "M"]
    modularAnalysis.variablesToNtuple("B-", B_vars, filename=B_file, path=path)

    return path

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

A basf2 Example

```
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)
```

And now just execute the task chain by running the script with python, e.g. with:

```
python3 basf2_chain_example.py
```

Successful Execution

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

Scheduled 20 tasks of which:
* 17 complete ones were encountered:
    - 1 AnalysisEvaluationTask(...)
    - 1 EfficiencyEvaluationTask(...)
    - 14 ResolutionCorrectionTask(...)
    - 1 ResolutionCorrectionValidationTask(...)
* 3 ran successfully:
    - 1 OfflineAnalysisMaster(...)
    - 1 SidebandEvaluationTask(...)
    - 1 ValidationWrapperTask(...)

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

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

Unsuccessful Execution

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

Scheduled 50 tasks of which:
* 16 complete ones were encountered:
    - 1 AnalysisEvaluationTask(...)
    - 1 EfficiencyEvaluationTask(...)
    - 14 SampleCombinerTask(...)
* 3 ran successfully:
    - 1 WeightAnalysisTask(...)
    - 2 WeightFinalizationTask(...)
* 12 failed:
    - 12 WeightFinalizationTask(...)
* 14 were left pending, among these:
    * 14 had failed dependencies:
        - 14 ResolutionCorrectionTask(...)

This progress looks :( because there were failed tasks

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

If Tasks **fail** for some reason

- the error message will be in the log output when run locally, or
- B2Luigi will point you to the log file of the respective job, when running in batch mode.

Further Development

Submitting Basf2PathTask to the grid is not supported, yet!

. . . but **Michael Eliachevitch**[2] is working on a *soft* integration of gbasf2 into B2Luigi:

- *Soft*, because the Task itself will not run on the grid;
- It will rather submit to the grid via gbasf2 and wait for the jobs to finish.
- For this it will submit a file containing the pickled basf2 path.
- This way you can make use of extensive external packages to create the basf2 path without shipping the package with the grid job.

The details are not clear at the moment, but he is working on it. . .

[2] meliache@uni-bonn.de

Availability of Luigi and B2Luigi

Luigi and B2Luigi are available in the current basf2 externals `v01-08-00`**:**

- luigi 2.7.7
- b2luigi 0.3.2

These versions already support most of the features shown in this talk.

HTCondor support is available since **b2luigi version 0.4.0**

How to install newer B2Luigi versions:

```
pip3 install [--user] --ignore-installed b2luigi==0.4.4
```

See also the information given on the documentation for B2Luigi developers!

B2Luigi Experts

Main Developer

- Nils Braun nilslennartbraun@gmail.com

Power Users

- Felix Metzner felix.metzner@kit.edu
- Michael Eliachevitch meliache@uni-bonn.de
- Maximilian Welsch mwelsch@uni-bonn.de
- Patrick Ecker patrick.ecker@student.kit.edu

Feel free to contact any of us, if you encounter any issues or have general questions!

General Advise

Keep the structure of

- **the** `output` **and** `input` **methods**
- **and the task logic**

the same. . .

Summary

Luigi provides a **well tested and documented** framework to manage data processing workflows.
It is used by **many organizations and companies** to handle their data processing.

B2Luigi provides additional features to take care of some housekeeping for you and adds functionality to make use of our **batch systems** as well as to make the implementation of **basf2 Tasks** as easy as possible for you.

Together, these tools can make automatization easy and thus can ensure reproducibility of your work!

Thank You for Your attention!