ceci Documentation

Release 0.0.1

Joe Zuntz

May 11, 2023

Contents:

1 Installation 3

2 Overview 5

3 Tutorial 7

4 Pipeline Stages 9

5 Pipeline YAML Files 13

6 Launchers 17

7 Sites 19

8 Indices and tables 21

Ceci is a framework for defining and running DESC pipelines under the Parsl workflow management system. This means it connects together individual tasks that depend on each other's outputs and runs them, potentially in parallel, passing the outputs of one onto the next.

CHAPTER 1

Installation

Ceci requires python 3. It can be installed using pip:

```
pip install ceci
```

Or from source like this:

```
git clone https://github.com/LSSTDESC/ceci
cd ceci
python3 setup.py install
```

CHAPTER 2

Overview

Ceci lets you define and run pipelines - sequences of calculation steps that can depend on earlier steps - and run them under the parsl workflow system (and perhaps in future other systems).

In the ceci model each step in the calculation is defined by writing a python class implementing particular pre-defined methods.

Then you actually run your pipeline by running the ceci command on a configuration file in the YAML format.

CHAPTER 3

Tutorial

First, install ceci by following the instructions on the Installation page.

To run the test example you'll need to use the source code

3.1 Running a test pipeline

A mock pipeline, which just reads from and writes to a series of small text files, can be run by with the command:

```
ceci tests/test.yml
```

3.2 Making a new pipeline

You can use a cookiecutter template to make new pipeline stages. You can install cookiecutter with `pip3 install cookiecutter` and then run:

```
cookiecutter https://github.com/LSSTDESC/pipeline-package-template
```

And enter a name for your pipeline collection.

This will create a template for your new pipeline stages. You design your pipeline stages in python files in this new repo - the example in `<repo_name>/<repo_name>stage1.py` shows a template for this, and you can see the "Stages" section for more details.

Your job as a pipeline builder is to make a file like this for each stage in your pipeline, and fill them in. You can then

CHAPTER 4

Pipeline Stages

4.1 Overview

A PipelineStage implements a single calculation step within a wider pipeline. Each different type of analysis stge is represented by a subclass of PipelineStage.

The base class handles the connection between different pipeline stages, and the execution of the stages within a workflow system (parsl or cwl), potentially in parallel (MPI).

The subclasses must:

- define their name
- define their inputs and outputs
- provide a "run" method which does the actual execution of the pipeline step.

They must use base class methods within the run method to find their input and output file paths. They can optionally use further methods in this class to open and prepare those files too.

4.2 Inputs/Outputs and Tags

The I/O system for Ceci uses the concept of "tags". A tag is a string which corresponds to a single input or output file. Using it allows us to easily connect together pipeline stages by matching output tags from earlier stages to input tags for later ones. Tags must be unique across a pipeline.

4.3 Configuration Parameters

Every stage has an additional implicit input: a configuration file in YAML format.

Stage classes can define a dictionary as an instance variable listing what variables it needs in that configuration file and their types, or give default values in case they are not found in the parameter file.

Here is an example:

```
class MyStage:
    ...
    config_options = {
        'T_cut':float,
        's2n_cut':float,
        'delta_gamma': float,
        'max_rows':0,
        'chunk_rows':10000,
        'zbin_edges':[float]
    }
```

Some parameters like `T_cut` have been given just a python type, indicating that there is no default value for them and the use should specify a value of type "float" in the parameter file. Others like `max_rows` have a default value that will be used if the parameter is not otherwise specified.

The parameter file will automatically be read and the results put into a dictionary that the stage can access via `self.config`, for example: `cut = self.config['T_cut']`.

More complicated parameter types such as dictionaries can also be used, but they cannot currently be specified in the `config_option` dictionary and so the system will not automatically check for their presence in the parameter file - you will have to do that yourself.

Parameters can also be overridden when running a stage on its own on the command line, (see "Execution", below) by using them as flag: `--T_cut=0.4`

4.4 Pipeline Methods

The full set of pipeline methods is documented below. Of particular note are the methods described here, which are designed to be used by subclasses.

Return the path to input or output files:

```
self.get_input(tag)
self.get_output(tag)
```

Get the base class to find and open an input or output file for you, optionally returning a wrapper class instead of the file:

```
self.open_input(tag, wrapper=False, **kwargs)
self.open_output(tag, wrapper=False, **kwargs)
```

Look for a section in a yaml input file tagged "config" and read it. If the config_options class variable exists in the class then it checks those options are set or uses any supplied defaults.

```
self.get_config()
```

MPI attributes for parallelization.

```
self.rank
self.size
self.comm
```

If the code is not being run in parallel, comm will be None, rank will be 0, and size will be 1.

IO tools - reading data in chunks, splitting up according to MPI rank, if used

```
self.iterate_fits(tag, hdunum, cols, chunk_rows)
self.iterate_hdf(tag, group_name, cols, chunk_rows)
```

4.5 Execution

Pipeline stages can be automatically run as part of a pipeline, or manually run on the command line, using the syntax:

```
python </path/to/pipeline_implementation.py> <StageName> --<input_name1>=</path/to/
↪input1.dat>
    --<input_name2>=</path/to/input2.dat>  --<output_name1>=</path/to/output1.dat>
```

4.6 API

The complete pipeline stage API is below - stages not described above are mostly used internally by the pipeline system.

CHAPTER 5

Pipeline YAML Files

Two YAML-format configuration files are needed to run a pipeline.

The first describes which steps to run in a pipeline, the overall inputs for it, execution information, and directories for outputs. It is described on this page. It includes the path to the second file, (see *Config* below); that file is described in more depth on the page config2.

Here is an example, from `test/test.yml`. The different pieces are described below.

```
# There are currently three defined launchers
# mini, parsl, and cwl
launcher:
    name: mini
    interval: 0.5
# and three sites:
# local, cori, and cori-interactive
site:
    name: local
    max_threads: 2


# The list of stages to run and the number of processors
# to use for each.
stages:
    - name: WLGCSummaryStatistic
      nprocess: 1
    - name: SysMapMaker
      nprocess: 1
    - name: shearMeasurementPipe
      nprocess: 1
    - name: PZEstimationPipe
      nprocess: 1
    - name: WLGCRandoms
      nprocess: 1
    - name: WLGCSelector
      nprocess: 1
```

(continues on next page)

(continued from previous page)

```
    - name: SourceSummarizer
      nprocess: 1
    - name: WLGCTwoPoint
      nprocess: 1
    - name: WLGCCov
      nprocess: 1

# Definitions of where to find inputs for the overall pipeline.
# Any input required by a pipeline stage that is not generated by
# a previous stage must be defined here.  They are listed by tag.
inputs:
    DM: ./test/inputs/dm.txt
    fiducial_cosmology: ./test/inputs/fiducial_cosmology.txt

# Overall configuration file
config: ./test/config.yml

# If all the outputs for a stage already exist then do not re-run that stage
resume: False

# Put all the output files in this directory:
output_dir: ./test/outputs

# Put the logs from the individual stages in this directory:
log_dir: ./test/logs

# These will be run before and after the pipeline respectively
pre_script: ""
post_script: ""
```

5.1 Modules

The `modules` option, which is a string, consists of the names of python modules to import and search for pipeline stages (with spaces between each).

Each module is imported at the start of the pipeline. For a stage to be found, it should be imported somewhere in the chain of imports under `__init__.py` in one of the packages listed here. You can specify subpackages, like `module.submodule` in this list after `module` if you need to.

The `python_paths` option can be set to a single string or list of strings, and gives paths to add to python's `sys.path` before attempting the import above.

5.2 Stages

The `stages` parameter should be a list of dictionaries. Each element in the list is one pipeline stage to be run. You don't have to put the stages in order - ceci will figure that out for you.

Each dictionary represents one stage, and has these options, with the defaults as shown:

```
- name: NameOfClass       # required
  nprocess: 1             # optional
  threads_per_process: 1  # optional
  nodes: 1                # optional
```

`threads_per_process` is the number of threads, and therefore also the number of cores to assign to each process. OpenMP is the usual threading method used for our jobs, so `OMP_NUM_THREADS` is set to this value for the job.

`nodes` is the number of nodes to assign to the job. The processes are spread evenly across nodes.

`nprocess` is the total number of processes, (across all nodes, not per-node). Process-level parallelism is currently implemented only using MPI, but if you need other approaches please open an issue.

5.3 Launcher

The `launcher` parameter should be a dictionary that configures the workflow manager used to launch the jobs.

The `name` item in the dictionary sets which launcher is used. These options are currently allowed: `mini`, `parsl`, and `cwl`.

See the *Launchers* page for information on these launchers, and the other options they take.

5.4 Site

The `site` parameter should be a dictionary that configures the machine on which you are running the pipeline.

The `name` item in the dictionary sets which site is used. These options are currently allowed: `local`, `cori-batch`, and `cori-interactive`.

See the *Sites* page for information on these sites, and the other options they take.

5.5 Inputs

The `inputs` parameter is required, and should be set to a dictionary. It must describe any files that are overall inputs to the pipeline, and are not generated internally by it. Files that are made inside the pipeline must not be listed.

The keys are tags, strings from the `inputs` attribute on the classes that represent the pipeline stage. They should map to values which are the paths to find those inputs.

5.6 Config

The parameter `config` is required, and should be set to a path to another input YAML config file.

See the config2 page for what that file should contain.

5.7 Resume

The parameter `resume` is required, and should be set to `True` or `False`.

If the parameter is `True`, then any pipeline stages whose outputs all exist already will be skipped and not run.

In the current implementation, a pipeline stage with missing input will not cause "downstream" stages to be run as well - e.g. if the final stage in your pipeline has all its outputs present it will *not* be re-run, even if earlier stages *are* re-run because their outputs had been removed.

5.8 Directories

The parameter `output_dir` is required, and should be set to a directory where all the outputs from the pipeline will be saved. If the directory does not exist it will be created.

If the resume parameter is set to True, then this is the directory that will be checked for existing outputs.

The parameter `log_dir` is required, and should be set to a directory where the printed output of the stages will be saved, in one file per stage.

5.9 Scripts

Two parameters can be set to run additional scripts before or after a pipeline stage. You can use them to perform checks or process results.

Any executable specified by `pre_script` will be run before the pipeline. If it returns a non-zero status then the pipeline will not be run and an exception will be raised.

Any executable specified by `post_script` will be run after the pipeline, but only if the pipeline completes successfully. If the post_script returns a non-zero status then it will be returned as the ceci exit code, but no exception will be raised.

Both scripts are called with the same arguments as the original executable was called with.

CHAPTER 6

Launchers

Launchers are the system that actually runs a pipeline, launching and monitoring jobs, checking output, etc.

There are currently three launchers supported by Ceci, `mini`, `parsl`, and `cwl`, but it's easy for us to add more - please open an issue if you need this.

See also the *Sites* page for how to configure other aspects of where the pipeline is run - different launchers support different site options.

6.1 Minirunner

The `mini` launcher is a minimal in-built launcher with only basic features, but it's useful for small to medium sized jobs.

Minirunner understands the concept of Nodes versus Cores on supercomputers, and on Cori the numbers are determined from SLURM environment variables. If running on the login node, one node with four cores is assigned.

Minirunner does not launch jobs - if you want to use it in Cori batch mode you should call it from within the job submission script.

6.1.1 Minirunner options

The minirunner has one option, which is common to all sites:

```
launcher:
    name: mini
    interval: 3       # optional
```

`interval` is optional and controls number of seconds between checks that each stage is complete. It defaults to three seconds.

6.2 Parsl

Parsl is a fully-featured workflow manager. It can be configured for a very wide variety of machines and systems. It knows how to submit jobs using SLURM and other systems.

6.2.1 Parsl options

Parsl has one option, which is common to all sites:

```
launcher:
    name: parsl
    log: ""       # optional
```

`log` chooses a file in which to put overall top-level parsl output, describing the monitoring of jobs and output.

6.3 CWL

Common Workflow Language is a general language for describing workflows, that can be imported by multiple workflow engines. A reference implementation called `cwltool` can be used locally to run CWL pipelines.

6.3.1 CWL options

CWL has one option, which is common to all sites:

```
launcher:
    name: cwl
    dir: <path>       # required
    launch: cwltool   # optional
```

`dir` controls the directory where the CWL files describing the pipeline and the individual jobs are saved. If it does not exist it will be created.

`launch` controls the executable run on the CWL files. The default *cwltool* is actually expanded to `cwltool --outdir {output_dir} --preserve-environment PYTHONPATH`.

CHAPTER 7

Sites

A site is a machine where a pipeline is to be run. Ceci currently only supports running a pipeline at a single site, not splitting it up between them.

Three sites are currently supported: `local`, `cori-batch`, and `cori-interactive`.

See also the *Launchers* page for how to configure the manager that runs the pipeline.

7.1 Common Options

All sites have these global options:

```
site:
    name: local
    mpi_command: mpirun -n    # optional
    image: ""                 # optional
    volume ""                 # optional
```

`mpi_command` sets the name of the command used to launch MPI jobs. Its default depends the site.

`image` sets the name of a docker/shifter container in which to run jobs. It defaults to None, meaning not to use a container.

`volume` sets an option to pass to docker/shifter to mount a directory inside the container. It takes the form */path/on/real/machine:/path/inside/container*

7.2 Local

The local site is a general one and represents running in a straightforward local environment. Jobs are run using the python `subprocess` module.

```
site:
    name: local
    max_threads: 2    # optional
```

max_threads is optional and controls the maximum number of stages run at the same time. Its default depends on the launcher used.

7.3 Cori Interactive

The cori-interactive site is used to run jobs interactively on NERSC compute nodes. You should first use the salloc command to get an interactive allocation, and then within that run ceci.

There are no additional options for the cori-interactive site: the number of parallel stages is given by the number of nodes that you ask for in salloc.

7.4 Cori Batch

The site cori-batch runs on the Cori supercomputer at NERSC, and submits jobs to the SLURM batch system. In this mode, you should call ceci from the login node and stay logged in while the jobs run.

These options can be used for the cori-batch site:

```
launcher:
    name: cori
    cpu_type: haswell    # optional
    queue: debug         # optional
    max_jobs: 2          # optional
    account: m1727       # optional
    walltime: 00:30:00   # optional
    setup:  /global/projecta/projectdirs/lsst/groups/WL/users/zuntz/setup-cori
    # ^^ optional
```

cpu_type is optional and controls which partition of cori is used for jobs, and should be *haswell* or *KNL*.

queue is optional and controls which SLURM queue jobs are launcher on. It can be debug, regular, or premium. See the n
for a description of each.

max_jobs is optional and controls the maximum number of SLURM jobs submitted using sbatch.

account is optional and controls the name of the account to which to charge SLURM jobs. You need to be a member of the associated project to use an account.

walltime is optional and controls the amount of time allocated to each SLURM job.

setup is optional and selects a script to be run at the start of each SLURM job.

CHAPTER 8

Indices and tables

- genindex
- modindex
- search