Operon Documentation

Release 0.1.0

Dominic Fitzgerald

Mar 20, 2018

Contents

CHAPTER 1

Getting Started

To install with pip:

```
$ pip install operon
```

Operon keeps track of pipelines and configurations automatically in a hidden directory. Before Operon can be used, this directory needs to be initialized with the command:

```
$ operon init
```

To install a pipeline:

```
$ operon install /path/to/pipeline.py
```

To configure the pipeline and optionally install software:

```
$ operon configure <pipeline-name>
```

To run the pipeline:

```
$ operon run <pipeline-name>
```

CHAPTER 2

Using Operon

The primary user interaction with Operon is through the command line interface, exposed as the executable `operon`. Subcommands are provided to install, manage, and run pipelines compatible with the Operon framework.

2.1 Initialization

Operon keeps track of pipelines and configurations, among other metadata, in a hidden directory. Before Operon can be used, this directory structure needs to be initialized:

```
$ operon init [init-path]
```

Where `init-path` is an optional path pointing to the location where the hidden `.operon` folder should be initialized. If this path isn't given it will default to the user's home directory.

By default, whenever Operon needs to access the `.operon` folder it will look in the user's home directory. If the `.operon` folder has been initialized elsewhere, there must be a shell environment variable `OPERON_HOME` which points to the directory containing the `.operon` folder.

Note: If a path other than the user's home directory is given to the `init` subprogram, it will attempt to add the `OPERON_HOME` environment variable to the user's shell session in `~/.bashrc`, `~/.bash_profile`, or `~/.profile`.

After a successful initialization, a new shell session should be started for tab-completion and the `OPERON_HOME` environment variable to take effect.

2.2 Pipeline Installation

To install an Operon compatible pipeline into the Operon system:

```
$ operon install /path/to/pipeline.py [-y]
```

The pipeline file will be copied into the Operon system and optionally Python package dependencies, as specified by the pipeline just installed, can be installed into the current Python environment using `pip`.

> **Caution:** If `install` attempts to install Python package dependencies, it will attempt to do so using the `--upgrade` flag to `pip`. If in the current Python environment those packages already exist, they will be either upgraded or downgraded, which may cause other software to stop functioning properly.

2.3 Pipeline Configuration

To configure an Operon pipeline with platform-static values and optionally use Miniconda to install software executables that the pipeline uses:

```
$ operon configure <pipeline-name> [-h] [--location LOCATION] [--blank]
```

If this is the first time the pipeline has been configured and Miniconda is found in `PATH`, then the `configure` subprogram will attempt to create a new conda environment, install software instances that the pipeline uses, then inject those software paths into the next configuration step. If a conda environment for this pipeline has been created before, `configure` can attempt to inject those software paths instead.

For the configuration step, Operon will ask the user to provide values for the pipeline which will not change from run to run such as software paths, paths to reference files, etc. The question is followed by a value in brackets (`[]`), which is the used value if no input is provided. If a conda environment is used, this value in brackets will be the injected software path.

By default, the configuration file is written into the `.operon` folder where it will automatically be called up when the user runs `operon run`. If `--location` is given as a path, the configuration file will be written out there instead.

2.4 Seeing Pipeline States

To see all pipelines in the Operon system and whether each has a corresponding configuration file:

```
$ operon list
```

To see detailed information about a particular pipeline, such as current configuration, command line options, any required dependencies, etc:

```
$ operon show <pipeline-name>
```

2.5 Run a Pipeline

To run an installed pipeline:

```
$ operon run <pipeline-name> [--pipeline-config CONFIG] [--parsl-config CONFIG] \
                             [--logs-dir DIR] [pipeline-options]
```

The set of accepted `pipeline-options` is defined by the pipeline itself and are meant to be values that change from run to run, such as input files, metadata, etc. Three options will always exist:

- `pipeline-config` can point to a pipeline config to use for this run only
- `parsl-config` can point to a file containing JSON that represents a Parsl config to use for this run only
- `logs-dir` can point to a location where log files from this run should be deposited; if it doesn't exist, it will be created; defaults to the currect directory

When an Operon pipeline is run, under the hood it creates a Parsl workflow which can be run in many different ways depending on the accompanying Parl configuration. This means that while the definition for a pipeline run with the `run` subprogram is consistent, that actual execution model may vary if the Parsl configuration varies.

2.5.1 Parsl Configuration

Parsl is the package the powers Operon and and is responsible for Operon's powerful and flexible parallel execution. Operon itself is only a front-end abstraction of a Parsl workflow; the actual execution model is fully Parsl-specific and as such it's advised to check out the Parsl documentation to get a sense for how to design a Parls configuration for a specific need-case.

The `run` subprogram attempts to pull a Parsl configuration from the user in the following order:

1. From the command line argument `--parsl-config`
2. From the pipeline configuration key `parsl_config`
3. From a platform default JSON file located at `$OPERON_HOME/.operon/parsl_config.json`
4. A default parsl configuration provided by the pipeline
5. A package default parsl configuration of 8 workers using Python threads

For more detailed information, refer to AnotherPage

2.6 Command Line Help

All subcommands can be followed by a `-h`, `--help`, or `help` to get a more detailed explanation for how it should be used.

CHAPTER 3

Building Pipelines

3.1 Dependency Workflow Concepts

Operon pipelines don't actually run anything at all; instead, they define a workflow. Parsl then uses the defined workflow to run the pipeline in a highly-efficient and configuration manner. Operon simply acts as an abstraction to Parsl so the developer need only what about what software needs to be run, and at what point in the pipeline it has all the information it needs to run successfully.

There are two main components to a dependency workflow graph:

1. Executables
2. Data

Executables generally take input, perform some computational work on that input, then produce some output. Data is any file on disk that is used as input or produced as output and that needs to be considered in the workflow graph.

When building a pipeline in Operon, the connections between Executables and Data need not be a part of the design process. The developer needs only to define Executables which are a part of the workflow, and the input and/or output files for each Executable. At runtime, Operon will examine the dependency workflow graph and feed the connections appropriately into Parsl.

As an example, consider the following scenario in a bioinformatics workflow : a FASTQ file is used as input to an aligner (say, bwa), which produces a BAM file. That produced BAM file needs to be used as input to two programs, one to gather flagstats and one to quantify gene counts. The dependency workflow graph would look as follows:

The developer for this pipeline only needs to define the following:

- There's a program called bwa, which lives at `/path/to/bwa`. As input, it takes in a file located at `/path/to/fastq`, and as output it generates a file called `/path/to/bam`.

- There's a program called samtools, which lives at `/path/to/samtools`. As input, it takes in a file located at `/path/to/bam`, and as output it generates a file called `/path/to/flagstats`.
- There's a program called genecounter, which lives at `/path/to/genecounter`. As input, it takes in a file located at `/path/to/bam`, and as output it generates a file called `/path/to/genecounts`.

This defines three Software instances (Apps in Parsl verbage): bwa, samtools, and genecounter. All three Software instances have data dependencies; that is, they are require input data to run. However, two of the Software instances' data dependencies are not yet available (because they haven't been produced by the pipeline yet), so they will not run until those dependencies become available. The Software bwa, however, has all of its data dependencies available, so it begins running immediately. Once bwa is finished running, and consequently produces its output `/path/to/bam`, the Software samtools and genecounter both recognize that their data dependencies are now available, and so both begin running concurrently.

Note: `/path/to/fastq`, `/path/to/bwa`, etc are placeholders in the above example. In a real pipeline, the developer would gather those values either from the command line via `pipeline_args` or from the configuration via `pipeline_config`. As long as `Data()` inputs and outputs resolve to filesystem paths at the time of workflow generation, Parsl will be able to correctly determine data dependencies.

3.2 Pipeline Meta Definitions

Pipeline meta definitions describe how the pipeline should be installed, provisioned, and configured so that as little as possible needs to be done by the user before the pipeline is ready to run on the user's platform.

All pipeline meta definitions (and logic, for that matter) is defined in a single document with a single class, always called `Pipeline`, which subclasses `operon.components.ParslPipeline`.

```python
from operon.components import ParslPipeline

class Pipeline(ParslPipeline):
    def description(self):
        return 'An example pipeline'

    ...

    def pipeline(self, pipeline_args, pipeline_config):
        # Pipeline logic here
```

3.2.1 Description

The description of the pipeline is a string meant to be a human readable overview of what the pipeline does and any other relevant information for the user.

```python
def description(self):
    return 'An example pipeline, written in Operon, powered by Parsl'
```

The pipeline description is displayed when the user runs `operon show`.

3.2.2 Dependencies

Pipeline dependencies are Python packages which the pipeline logic use. Dependencies are provided as a list of strings, where each string is the name of a package available on PyPI and suitable to be feed directly into `pip`.

```
def dependencies(self):
    return [
        'pysam==0.13',
        'pyvcf'
    ]
```

Upon pipeline installation, the user is given the option to use `pip` to install dependencies into their current Python environment. While this may be convenient, it may also cause package collisions or unecessary muddying of a distribution Python environment, so the user can instead opt to get the dependencies from `operon show` and install them manually into a Python virtual environment.

Note: If the user accepts auto-installing dependencies into their current Python environment, `pip` will attempt to do so using the `--upgrade` flag. This may upgrade or downgrade packages already installed in the current Python environment if there are any collisions.

3.2.3 Conda/Bioconda

Executables provided by Conda/Bioconda can be installed and injected into the user's pipeline configuration, provided the user has Miniconda installed and in PATH. Executables are defined by a list of `CondaPackage` tuples, with the option to override the default conda channels that Operon loads.

```
from operon.components import CondaPackage

def conda(self):
    return {
        'channels': ['overriding', 'channels', 'here'],
        'packages': [
            CondaPackage(tag='star=2.4.2a', config_key='STAR', executable_path='bin/
↪STAR'),
            CondaPackage(tag='picard', config_key='picard', executable_path='share/
↪picard-2.15.0-0/picard.jar')
        ]
    }
```

If provided, `channels` will be loaded by Miniconda in list order, which means the last entry has the highest precedence, the second-highest entry has the second-highest precedence, etc.

A `CondaPackage` named tuple takes the following keys:

- `tag` is the name of the executable and optional version number fed directly to Miniconda
- `config_key` is the outermost key in the pipeline's `configuration()`. When this executable is injected into a user's pipeline config, it's placed into `pipeline_config[config_key]['path']`
- `executable_path` is only necessary if the basename of the installed executable is different from the conda tag, or if the developer wishes to use an executable outside conda's default `bin` folder. Some examples:
  - The conda package `star=2.4.2a` is installed as `STAR`, so `executable_path=` must be set to `bin/STAR`
  - The conda package `picard` installs an executable into `bin`, but if the developer wishes to access the jar file directly, she must set `executable_path=` to `share/picard-2.15.0-0/picard.jar`
  - The conda package `bwa` installs an executable into `bin` called `bwa`, so `executable_path` does not need to be set

To see which executables are offered by Bioconda, please refer to their package index.

3.2.4 Parsl Configuration

A default Parsl configuration can be provided in the event the user doesn't provide any higher-precendence Parsl configuration. The returned `dict` will be fed directly to Parsl before execution.

```
def parsl_configuration(self):
    return {
        'sites': [
            {
                'site': 'Local_Threads',
                'auth': {'channel': None},
                'execution': {
                    'executor': 'threads',
                    'provider': None,
                    'max_workers': 4
                }
            }
        ],
        'globals': {'lazyErrors': True}
    }
```

To better understand Parsl configuration, please refer to their documentation on the subject.

Note: This method of configuring Parsl has very low precedence, and that's on purpose. The user is given every opportunity to provide a configuration that works for her specific platform, so the configuration provided by the pipeline is only meant as a desperation-style "we don't have anything else" configuration.

3.2.5 Pipeline Configuration

The pipeline configuration contains attributes passed into the pipeline logic which may change from platform to platform, but generally won't change from run to run. For example, paths to executables for software, paths to reference files, number of threads to use, etc will vary by platform but will be the same for every run.

```
def configuration(self):
    return {
        'software1': {
            'path': 'Full path to software1',
            'threads': 'Run software1 with this many threads',
            'threads': {
                'q_type': 'list',
                'message': 'Run software1 with this many threads',
                'choices': ['1', '2', '4', '8', '16'],
                'default': '4'
            }
        },
        'software2': {
            'path': 'Full path to software2',
            'genome_reference': {
                'q_type': 'path',
                'message': 'Path to genome reference'
            }
        }
    }
```

The returned configuration dictionary may nest arbitrarily deep. All values must be either a dictionary or a string. Considering the configuration dictionary as a tree, there are two types of leaves: a string or a dictionary which configures a question to the user. During configuration of the pipeline using `operon configure`, the user is presented with a prompt for each leaf, and the user input is gathered and stored in place of the prompt string.

Note: The nesting of dictionaries inside the configuration dictionary is purely for the developer's organizational convenience; the user will never see anything but prompts defined by the string values.

If the order of prompts is important, return a `collections.OrderedDict` instance.

For a string leaf, the question type defaults to a Text question, where the prompt presented is the string itself. The exception to this is if the word `path` is found in the most immediate key, the question type will default to `Path`.

For a dictionary leaf, the question type can be fully configured. For a dictionary to be recognized as a leaf, it must contain the key `q_type`, or else it will be interpreted as another level in the nested configuration dictionary. The following options can be passed to a question configuration:

- `q_type` must be one of `{path, text, confirm, list, checkbox, password}`
- `message` is the prompt displayed to the user
- `default` is a default value suggested to the user as part of the prompt
- `validate` is a function which determines whether the user input is valid
- `ignore` is a function which determines whether to display this question to the user
- `choices` is a list of choices; only used by the List and Checkbox question types
- `always_default` if present with any value, will force the default to always be the value defined by the key `default`, regardless of whether another value was injected by Operon

The `q_type` and `message` keys are required for all question types, while the `choices` key is additionally required for List and Checkbox question types. For more information on how each of the question types operate, please refer to the inquirer documentation on question types.

For the above example configuration, the user will see and interactively fill in the prompts:

```
$ operon configure pipeline-name
[?] Full path to software1: (User enters) /path/to/soft1
[?] Run software1 with this many threads: (User selects) 8
   1
   2
   4
 > 8
   16

[?] Full path to software2: (User enters) /path/to/soft2
[?] Path to genome reference: (User enters) /path/to/genome
```

The input from the user is stored in the `.operon` folder, so the next time the pipeline is run with this configuration it will be made available in the `pipeline_config` parameter:

```
# Contents of pipeline_config
{
    'software1': {
        'path': '/path/to/soft1',
        'threads': '8'
    },
    'software2': {
        'path': '/path/to/soft2',
        'genome_reference': '/path/to/genome'
    }
}
```

So for any software which needs access to a genome reference, the path can be passed to the software as `pipeline_config['software2']['genome_reference']`.

3.2.6 Pipeline Arguments

The pipeline arguments are attributes that will change from run to run and are specified by the user as command line arguments on a per-run basis. Pipeline arguments are added by modifying the `argparse.ArgumentParser` object passed into `self.arguments()`; refer to the documentation for `argparse` for futher details on how pipeline arguments can be gathered.

```python
def arguments(self, parser):
    parser.add_argument('--output-dir', help='Path to output directory')
    parser.add_argument('--fastqs', nargs='*', help='Paths to all input fastq files')
    parser.add_argument('--run-name', default='run001', help='Name of this run')
    # Nothing needs to be returned since parser is modified in place
```

Added arguments are exposed to the user when running `operon run`, according to the rules of the `argparse` module.

```
$ operon run pipeline -h
> operon run pipeline [-h] [--output-dir OUTPUT_DIR] [--fastqs [FASTQS [FASTQS ...]]]
>                     [--run-name RUN_NAME]
>
> Pipeline description is here
>
> optional arguments:
>   -h, --help            show this help message and exit
>   -c CONFIG, --config CONFIG
>                         Path to a config file to use for this run.
>   --output-dir OUTPUT_DIR
>                         Path to output directory
>   --fastqs [FASTQS [FASTQS ...]]
>                         Paths to all input fastq files
>   --run-name RUN_NAME   Name of this run
>
$ operon run pipeline --fastqs /path/to/fastq1.fq /path/to/fastq2.fq \
>                     --output-dir /path/to/output --run-name run005
```

Populated arguments are made available to the pipeline as a dictionary in the `pipeline_args` parameter:

```
# Contents for pipeline_args
{
    'fastqs': ['/path/to/fastq1.fq', '/path/to/fastq2.fq'],
    'output_dir': '/path/to/output',
    'run_name': 'run005'
}
```

Note: Parameters in `argparse` can have dashes in them (and should to separate words), but when converted to a Python dictionary dashes are replaced with underscores.

Ex. `--output-dir` is accessed by `pipeline_args['output_dir']`

Three pipeline arguments are always injected by Operon: `--pipeline-config`, `--parsl-config`, and `--logs-dir`. These arguments point to a pipeline config file to use for the run, a Parsl config file to use for the run, and a directory in which to store log files, respectively.

3.3 Pipeline Logic

Pipeline logic defines how the workflow dependency graph should be built. The work is done in the `pipeline()` method, which is given two parameters, `pipeline_args` and `pipeline_config`, which are populated at runtime with command line arguments from the user and the stored pipeline configuration file, respectively.

Executables and data are defined using a set of wrapper objects provided by Operon: this section details those components and how to use them.

```
def pipeline(self, pipeline_args, pipeline_config):
    # All logic to build the workflow graph goes here
```

Note: All the logic here is only to build the workflow dependency graph, which means that *none of the executables are being run and none of the data is being produced until after* `pipeline()` *has completed*. Parsl only begins actually running the software after it's been fed the generated workflow graph.

All statements in the `pipeline()` method should be for generating the workflow graph, not handling or operating on data in any way. If needed, small blocks of Python can be written in a `CodeBlock` instance, which can be integrated into the workflow graph and so will execute at the correct time.

3.3.1 Data `operon.components.Data`

A `Data` instance wraps a file on the filesystem and registers it as a data node in the workflow graph. Any file that should be considered in the workflow graph needs to be wrapped in a `Data` instance; often this is only input or output to an executable, and may not include output like log files.

When passed as an argument to a `Parameter` object, the data must be specified as either input or output by calling either the `.as_input()` or `.as_output()` method. This distinction is not necessary when passing as a part of `extra_inputs=` or `extra_outputs=` keyword arguments.

`Data` objects can be marked as temporary, which designates the underlying file on the filesystem to be deleted at the end of the run, by setting the `tmp=` parameter to `True` in `.as_output()`.

```
from operon.components import Data

bwa.register(
    Parameter('--fastq', Data('/path/to/fastq.fq').as_input()),
    Parameter('--tmp-bam', Data('/path/to/tmp.bam').as_output(tmp=True)),
    Parameter('--persistent-bam', Data('/path/to/persistent.bam').as_output())
)

samtools.register(
    Parameter('--input-bam', Data('/path/to/persistent.bam').as_input())
)
```

The developer does not need to keep track of individual `Data` instances because `Data` instances are uniquely identified by the filesystem paths they wrap; that is, if a `Data` instance is created as `Data('/path/to/file')`, any subsequent calls to `Data('/path/to/file')` will not create a new `Data` instance but rather simply refer to the instance already created. Of course, the developer could store `Data` instances in variables and pass those instead, if desired.

`Data` instances can be used in-place anywhere a filesystem path would be passed; that includes both `Parameter` and `Redirect` objects.

3.3.2 Software `operon.components.Software`

A `Software` instance is an abstraction of an executable program external to the pipeline.

```
from operon.components import Software

bwa = Software(name='bwa', path='/path/to/bwa')
samtools_flagstat = Software(name='samtools', subprogram='flagstat')
genecounter = Software(name='genecounter', path='/path/to/genecounter')
```

If the `path=` parameter isn't given, Operon will try to infer the path by looking in `pipeline_config[name]['path']`. If the path can't be inferred, a `ValueError` will be thrown.

To register an Executable node in the workflow graph, call the `Software` instance's `.register()` method. `register()` takes any of `Parameter`, `Redirect`, `Pipe`. Keyword arguments `extra_inputs=` and `extra_outputs=` can also be given to pass in respective lists of `Data()` input and output that aren't defined as a command line argument to the Executable.

```
bwa.register(
    Parameter('--fastq', Data('/path/to/fastq.fq')),
    Parameter('--phred', '33'),
    Redirect(stream=Redirect.STDERR, dest='/logs/bwa.log'),
    extra_inputs=[Data('/path/to/indexed_genome.fa')],
    extra_outputs=[Data('/path/to/bam')]
)
```

The `register()` method returns an object wrapping the Executable node's id, which can be passed to other `Software` instances via the `wait_on=` keyword. If a `Software` is given other apps in its `wait_on=`, those other apps will be included in the input dependencies, and so won't start running until all app **and** data dependencies are resolved.

```
first_app = first.register(
    Parameter('-a', '1')
)

second.register(
    Parameter('--output', Data('second.out').as_output())
)

third.register(
    Parameter('b', Data('second.out').as_input()),
    wait_for=[first_app]
)
```

In the above example, `third` won't start running until both `first` is finished running and the output from `second` called `second.out` is available.

3.3.3 CodeBlock `operon.components.CodeBlock`

A `CodeBlock` instance wraps a Python function that can be passed `Data` instances in much the same way as a `Software` instance, and so can be integrated into the workflow graph. That is, a functions wrapped in a `CodeBlock` will wait to execute until all its data dependencies are available.

The function wrapped by a `CodeBlock` instance can be defined as normal and registered with `CodeBlock.register()`, where arguments and data dependencies can be defined.

```
def get_mapped_reads_from_flagstats(star_output_bam):
    import re
    with open(star_output_bam + '.flagstat') as flagstats:
        flagstats_contents = flagstats.read()
        target_line = re.search(r'(\d+) \+ \d+ mapped', flagstats_contents)
        if target_line is not None:
            with open('output.txt', 'w') as output_write:
                output_write.write(str(int(target_line.group(1))/2) + '\n')
CodeBlock.register(
    func=get_mapped_reads_from_flagstats,
    args=[],
    kwargs={'star_output_bam': star_output_bam},
    inputs=[Data(star_output_bam + '.flagstat').as_input()],
    outputs=[Data('output.txt').as_output()]
)
```

Note: When a function wrapped by a `CodeBlock` actually executes, the scope in which it was defined will be long gone. That means that any variables or data structures declared in `pipeline()` can't be counted on as available in the body of the function. It also means that any modules the function needs to use must be explicitly imported by the function, even if that module has already been imported by the pipeline document.

The return value of a `CodeBlock` is the same as that for a `Software` instance, and can be passed to other `Software` or `CodeBlocks` via the `wait_on=` keyword argument.

3.3.4 Parameter `operon.components.Parameter`

A `Parameter` object represents a parameter key and value(s) passed into a `Software` instance.

```
from operon.components import Parameter

Parameter('-a', '1')   # Becomes '-a 1'
Parameter('--type', 'gene', 'transcript')   # Becomes '--type gene transcript'
Parameter('--output=/path/to/output')   # Becomes '--output=/path/to/output'
```

When multiple `Parameter` instances are passed into a `Software` instance, order is preserved, which is important for positional arguments.

3.3.5 Redirect `operon.components.Redirect`

A `Redirect` objects represents an output stream redirection. The keyword arguments `stream=` and `dest=` direct which stream(s) to redirect and to where on the filesystem, respectively.

```
from operon.components import Redirect
```

```
bwa.register(
    Parameter('-a', '1000'),
    Redirect(stream=Redirect.STDOUT, dest='/path/to/bwa.log')
)
```

`stream=` can be one of the provided constants:

```
Redirect.STDOUT         # >
Redirect.STDOUT_APPEND  # >>
Redirect.STDERR         # 2>
Redirect.STDERR_APPEND  # 2>>
Redirect.BOTH           # &>
Redirect.BOTH_APPEND    # &>>
```

The order of `Redirect` objects passed to a `Software` instance, both in relation to each other and to other `Parameter` objects, doesn't matter. However, if more than two `Redirect` s are passed in, only the first two will be considered.

3.3.6 Pipe `operon.components.Pipe`

A `Pipe` object represents piping the output of one executable into the input of another. The producing `Software` instance is passed a `Pipe` object, which contains the receiving `Software` instance.

```
from operon.components import Pipe

software1.register(
    Parameter('-a', '1'),
    Pipe(software2.prep(
        Parameter('-b', '2'),
        Parameter('-c', '3')
    ))
)
# Registers as: software1 -a 1 | software2 -b 2 -c3
```

Note: Since the whole executable call needs to be registered with Parsl as a single unit, `register()` is only called on the outermost `Software` instances. Within a `Pipe` object, the receiving `Software` instance should instead call `prep()`, which takes all the same parameters as `register()`.

3.4 Pipeline Logging

All stream output from all Executables in the workflow graph that aren't explicitly redirected to a file with a `Redirect` is gathered and output to a single pipeline log file at the end of execution.

The location of this log file is defined by the user with the `--logs-dir` pipeline argument injected into every pipeline. It may be of interest to the developer to also put any explicitly redirected log files into this directory.

CHAPTER 4

Developer Tutorial

This tutorial is meant to guide you, the pipeline developer, through all the necessary concepts to write an optimally parallel and completely portable working pipeline in under 150 lines of code.

Since Operon was originally developed with bioinformatics in mind, we're going to be developing a bioinformatics pipeline and using bioinformatics concepts, but Operon certainly isn't limited to any single field. The developer can easily take the concepts talked about here and apply them to any set of software.

Let's get started. This tutorial assumes you have Operon installed. First we need a plain Python file; create one called `tutorial.py` in whatever editor you desire. The very first thing we need to do is create a class called `Pipeline` which is a subclass of `operon.components.ParslPipeline`. Add the following to `tutorial.py`:

```python
from operon.components import ParslPipeline


class Pipeline(ParslPipeline):
    def description(self):
        pass

    def dependencies(self):
        pass

    def conda(self):
        pass

    def arguments(self, parser):
        pass

    def configuration(self):
        pass

    def pipeline(self, pipeline_args, pipeline_config):
        pass
```

Pipelines are defined by overriding methods in `ParslPipeline` as above. We'll fill in those methods now, going from simple to more complex.

4.1 description(self)

The `description(self)` method needs only to return a string that is used to describe the pipeline on various help screens. Fill it in like so:

```
def description(self):
    return 'A tutorial pipeline'
```

4.2 dependencies(self)

The `dependencies(self)` method needs to return a list, where each element is a string representation of a Python packages acceptable to `pip`, which the user has the option to install at the time of pipeline installation. Generally these packages should be ones that are used in the pipeline but aren't a part of the standard library. In this tutorial, we're going to use `PyVCF` like so:

```
def dependencies(self):
    return ['pyvcf']
```

Note: When the user installs Python dependencies, they're installed into the current Python environment, which may or may not be the correct environment for eventual pipeline execution. There isn't much the developer can do about this, so don't stress too much.

4.3 conda(self)

The `conda(self` method is used to define what external software the pipeline uses, and returns a dictionary with two possible keys which are both optional:

```
{
    'packages': [
        # List of bioconda packages to install
    ],
    'channels': [
        # List of anaconda channels to use
    ]
}
```

Unless `channels` is provided, Operon will use bioconda to download packages. The elements of the `packages` key should be `operon.components.CondaPackage` objects (add that import now). We will be using the following:

```
def conda(self):
    return {
        'packages': [
            CondaPackage(tag='bwa', config_key='bwa'),
            CondaPackage(tag='macs2=2.1.1', config_key='macs2'),
            CondaPackage(tag='samtools=1.6', config_key='samtools'),
            CondaPackage(tag='picard=2.9.2', config_key='picard'),
            CondaPackage(tag='freebayes=1.1.0', config_key='freebayes')
        ]
    }
```

Note: The `conda(self)` method is not required but is a massive convenience to the user (which might also be you) because it enables the user to not have to track down and manually install software to run the pipeline. Everything defined here, which presumably encompasses all or most software used in the pipeline, can be automatically gathered and injected into the pipeline's configuration dictionary at the time of configuration.

4.4 arguments(self, parser)

For this simple tutoral, we'll only take three basic arguments like so:

```
def arguments(self, parser):
    parser.add_argument('--read', help='Path to read in fastq format')
    parser.add_argument('--output-dir', help='Path to an output directory')
    parser.add_argument('--lib', help='Name of this sample library')
```

4.5 configuration(self)

Most of our configuration will be paths, which is a common practice, with a threading question thrown in. Notice in `bwa|reference` and `freebayes|reference_fasta` the expanded leaf type is used so that we can get those as `path` questions instead of plain `text`, since we *are* asking for a path.

```
def configuration(self):
    return {
        'bwa': {
            'path': 'Path to bwa',
            'reference': {
                'q_type': 'path',
                'message': 'Path to a reference genome prefix for bwa'
            },
            'threads': 'Number of threads to run bwa'
        },
        'macs2': {
            'path': 'Path to macs2'
        },
        'samtools': {
            'path': 'Path to samtools'
        },
        'picard': {
            'path': 'Path to picard'
        },
        'freebayes': {
            'path': 'Path to freebayes',
            'reference_fasta': {
                'q_type': 'path',
                'message': 'Full path to reference fasta'
            }
        }
    }
```

4.6 pipeline(self, pipeline_args, pipeline_config)

Now the main part of the pipeline building process. We've defined our periphery and can assume that the parameters `pipeline_args` and `pipeline_config` have been populated and that all the software we've asked for is installed.

Generally the first step is to define `operon.components.Software` instances:

```
def pipeline(self, pipeline_args, pipeline_config):
    freebayes = Software('freebayes')
    bwa_mem = Software('bwa', subprogram='mem')
    macs2 = Software('macs2', subprogram='callpeak')
    picard_markduplicates = Software(
        name='picard_markduplicates',
        path=pipeline_config['picard']['path'],
        subprogram='MarkDuplicates'
    )
    samtools_flagstat = Software(
        name='samtools_flagstat',
        path=pipeline_config['samtools']['path'],
        subprogram='flagstat'
    )
    samtools_sort = Software(
        name='samtools_sort',
        path=pipeline_config['samtools']['path'],
        subprogram='sort'
    )
```

For `freebayes` the instantiation is very simple: the default path resolution is fine, and there isn't a subprogram to call. `bwa mem` and `macs2 callpeak` are slightly more involved, but only by adding a `subprogram=` keyword argument.

For `picard` and `samtools`, we're giving names that don't have a match in the configuration dictionary. That means the default path resoluation won't work, so we need to give it paths explicitly with the `path=` keyword argument.

Next some very simple pipeline setup, just creating the output directory where the user defined output to go. There may be more setup in more complicated pipelines. Add:

```
# Set up output directory
os.makedirs(pipeline_args['output_dir'], exist_ok=True)
```

Now we can start constructing our pipeline workflow. Modify the import statments at the top of the file to:

```
import os
from operon.components import ParslPipeline, CondaPackage, Software, Parameter,␣
↪Redirect, Data, CodeBlock
```

We'll need all those components eventually.

The general idea the developer should take when constructing the pipeline workflow is to think of software as stationary steps and data as elements flowing through those steps. Each stationary step has data coming in and data going out. With that mental model, we can construct the following workflow:

First we will run the software `bwa` whose output will flow into `samtools sort`; this will be sequential so there's no parallelization involved quite yet. The output of `samtools sort` will flow into both `samtools flagstat` and `picard markduplicates`, forming our first two-way branch. These two program will run in parallel the moment that `samtools sort` produces its output. From there, the output of `picard markduplicates` flows as input into both the `freebayes` variant caller and the `macs2` peak caller, forming another two-way branch. Finally, the output of `freebayes` will flow as input into a Python code block which uses `PyVCF`. The overall workflow will terminate when all leaves have completed; in this case, `samtools flagstat`, `macs2`, and the Python code block.

Let's dive in. The first software we need to insert into the workflow is `bwa`:

```
alignment_sam_filepath = os.path.join(pipeline_args['output_dir'], pipeline_args['lib
↪'] + '.sam')
bwa_mem.register(
    Parameter('-t', pipeline_config['bwa']['threads']),
    Parameter(pipeline_config['bwa']['reference']),
    Parameter(Data(pipeline_args['read']).as_input()),
    Redirect(stream='>', dest=Data(alignment_sam_filepath).as_output(tmp=True))
)
```

There's a lot going on here. First we define a filepath to send out alignment output from `bwa`. Then we call the `.register()` method of `bwa`, which signals to Operon that we want to insert `bwa` into the workflow as a stationary step; the input data and output data flow is defined in the arguments to `.register()`.

The first parameter is simple enough, just passing a `-t` in with the number of threads coming from our pipeline configuration. The second parameter is a positional argument pointing to the reference genome we wish to use for this alignment.

The third argument is importantly different. It's another positional argument mean to tell `bwa` where to find its input fastq file, but the path is wrapped in a call to a `Data` object. Using a `Data` object is how Operon knows which data/files on the filesystem should be considered as part of the workflow; failure to specify input or output paths inside `Data` objects will cause Operon to miss them and may result in workflow programs running before they have all their inputs! Notice also the `.as_input()` method is called on the `Data` object, which tells Operon not only is this path important, but it should be treated as input data into `bwa`.

Finally, the `Redirect` object (`bwa` send its output to `stdout`) sends the `stdout` stream to a filepath, again wrapped in a `Data` object and marked as output. The `tmp=True` keyword argument tell Operon to delete this file *after the whole pipeline is finished*, since we're not too interested in keeping that file around in our final results.

```
sorted_sam_filepath = os.path.join(pipeline_args['output_dir'], pipeline_args['lib']␣
↪+ '.sorted.sam')
samtools_sort.register(
    Parameter('-o', Data(sorted_sam_filepath).as_output()),
    Parameter(Data(alignment_sam_filepath).as_input())
)
```

The next step in the workflow is `samtools sort`, which takes the output from `bwa` as input and produces some output of its own. Notice again the important filepaths are wrapped in `Data` objects and given a specification as input or output.

Note: Although there is certainly a conceptual link between the output from `bwa` and the input here, that link does not need to be explicitly defined. As long as the same filepath is used, Operon will automatically recognize and link together input and output data flows between `bwa` and `samtools sort`.

After `samtools sort`, we're at our first intersection. It doesn't matter in which order we define the next steps, so add something similar to:

```
samtools_flagstat.register(
    Parameter(Data(sorted_sam_filepath).as_input()),
    Redirect(stream='>', dest=sorted_sam_filepath + '.flagstat')
)

dup_sorted_sam_filepath = os.path.join(pipeline_args['output_dir'], pipeline_args['lib
↪'] + '.dup.sorted.sam')
```

```
picard_markduplicates.register(
    Parameter('I={}'.format(sorted_sam_filepath)),
    Parameter('O={}'.format(dup_sorted_sam_filepath)),
    Parameter('M={}'.format(os.path.join(pipeline_args['logs_dir'], 'marked_dup_
↪metrics.txt'))),
    extra_inputs=[Data(sorted_sam_filepath)],
    extra_outputs=[Data(dup_sorted_sam_filepath)]
)
```

There are a couple things to notice here. `samtools flagstat` takes input in a `Data` object but its output is a plain string. This is because the output of `samtools flagstat` isn't used as an input into any other program, so there's no need to treat it as a special file.

Notice also that both `samtools flagstat` and `picard markduplicates` use the same filepath as input; in fact that's the point! Operon (really Parsl) will recognize that and make a parallel fork here, running each program at the same time. As the developer, you didn't have to explicitly say to fork here, it is just what the workflow calls for. This is what we mean by the workflow being automatically and optimally parallel.

Finally, there are some funny keyword arguments to `picard_markduplicates.register()`. This is because of how the parameters are passed to `picard markduplicates`: it wants the form `I=/some/path`, which mostly easily achieved with a format string, as we've done. But if we were to pass in like this:

```
Parameter('I={}'.format(Data(sorted_sam_filepath).as_input()))
```

that wouldn't work. Why not? It's how we've passed in special input data in the other programs!

When the `Data` object is coerced into a string, it just becomes its path as a plain string. If a `Data` object is used in a format string's `.format()` method, it will become it string representation before Operon can ever recognize it. To mitigate that, we can explicitly tell Operon what the special input and output `Data` items are in the `extra_inputs=` and `extra_outputs=` keyword arguments, respectively. Notice in those keyword arguments we don't need to call `.as_input()` or `.as_output()` because Operon can determine which is which from the keyword.

Now we come to our final two-way fork:

```
macs2_output_dir = os.path.join(pipeline_args['output_dir'], 'macs2')
os.makedirs(macs2_output_dir, exist_ok=True)
macs2.register(
    Parameter('--treatment', Data(dup_sorted_sam_filepath).as_input()),
    Parameter('--name', pipeline_args['lib']),
    Parameter('--outdir', macs2_output_dir),
    Parameter('--call-summits'),
    Parameter('--shift', '100')
)

vcf_output_path = os.path.join(pipeline_args['output_dir'], pipeline_args['lib'] + '.
↪vcf')
freebayes.register(
    Parameter('--fasta-reference', pipeline_config['freebayes']['reference_fasta']),
    Parameter(Data(dup_sorted_sam_filepath).as_input()),
    Redirect(stream='>', dest=Data(vcf_output_path).as_output())
)
```

There isn't much going on here that we haven't already discussed, so we won't go into detail. Both `macs2` and `freebayes` use the output of `picard markduplicates`, so they will run once the output of `picard markduplicates` is available.

Finally, the output of `freebayes` is used as input into a Python codeblock:

```
def get_first_five_positions(vcf_filepath, output_filepath):
    import vcf
    vcf_reader = vcf.Reader(open(vcf_filepath))
    with open(output_filepath, 'w') as output_file:
        for record in vcf_reader[:5]:
            output_file.write(record.POS + '\n')
CodeBlock.register(
    func=get_first_five_positions,
    kwargs={
        'vcf_filepath': vcf_output_path,
        'output_filepath': os.path.join(pipeline_args['output_dir'], 'first_five.txt')
    },
    inputs=[Data(vcf_output_path)]
)
```

`CodeBlocks` can be thought of as very similar to `Software` instances with the same data flow model.

4.7 Running the Tutorial Pipeline

The pipeline in complete! If you with to attempt to run it, simply install it into Operon, configure it (you might need to create a reference genome index), and run it with the small data bundle linked below.

Download the tutorial data bundle.

An awkward step in the Operon installation process is the necessity of a reference genome index. To generate one we need a reference genome `fasta` file and then to run `bwa index` over it; this mean that when we first configure our pipeline when it asks for the `bwa` reference genome prefix we won't have it, but we can come back and fill it in later.

```
$ /path/to/bwa index /path/to/downloaded/dm6.fa.gz
```

`bwa` dumps its index in the same folder as the genome fasta, so the reference prefix is just the path to the reference fasta.

Use the inluded *Drosophila melanogaster* DNA fastq as your `--read` input.

`freebayes` can't deal with a compressed reference genome fasta, so uncompress it before passing the path in the pipeline config.

CHAPTER 5

Operon API

CHAPTER 6

Changelog

6.1 v0.1.1 (released Jan 2018)

- If the `path=` argument isn't provided to a `Software` instance, the path will attempt to populate from `pipeline_config[software_name]['path']`
- Added `subprogram=` argument to `Software`
- Made tab completer program silent if it ever fails because it doesn't exist

6.2 v0.1.2 (released 21 Feb 2018)

- Better error messages when the pipeline developer doesn't override a method properly
- Better error messages when the pipeline configuration is malformed
- Added `parsl_config` key to all pipeline configurations by default
- Added > and 2> paths to the string representation of a Software's command
- Removed captured log output from screen; it still goes to the log file
- Added tags marking start time, end time, and duration of pipeline run to the log output
- Start and end time for each Software or CodeBlock added to log output
- Added "currently running" log entry, whenever the set of running programs changes
- Updated output for `operon list`
- Updated output for `operon show`
- When a conda environment already exists, added ability to reinstall
- Added `operon uninstall` to remove pipelines
- Refactored cleanup so that it always runs, even if some programs fail during the run

CHAPTER 7

Parsl

Operon is powered by Parsl. To get an understanding of how to most efficiently run Operon pipelines, check out the Parsl documentation.