Pydra - a flexible and lightweight dataflow engine for scientific analyses

Dorota Jarecka[‡*], Mathias Goncalves[¶‡], Christopher J. Markiewicz[¶], Oscar Esteban[¶], Nicole Lo[‡], Jakub Kaczmarzyk[§‡], Satrajit Ghosh[‡]

Abstract—This paper presents a new lightweight dataflow engine written in Python: *Pydra*. *Pydra* is developed as an open-source project in the neuroimaging community, but it is designed as a general-purpose dataflow engine to support any scientific domain. The paper describes the architecture of the software, as well as several useful features, that make *Pydra* a customizable and powerful dataflow engine. Two examples are presented to demonstrate the syntax and properties of the package.

Index Terms—dataflow engine, scientific workflows, reproducibility

Introduction

Scientific workflows often require sophisticated analyses that encompass a large collection of algorithms. The algorithms, that were originally not necessarily designed to work together, and were written by different authors. Some may be written in Python, while others might require calling external programs. It is a common practice to create semi-manual workflows that require the scientists to handle the files and interact with partial results from algorithms and external tools. This approach is conceptually simple and easy to implement, but the resulting workflow is often time consuming, error-prone and difficult to share with others. Consistency, reproducibility and scalability demand scientific workflows to be organized into fully automated pipelines. This was the motivation behind *Pydra* - a new dataflow engine written in Python, that is presented in this paper.

The *Pydra* package is a part of the second generation of the *Nipype* ecosystem ([GBM+11], [Dev]) --- an open-source framework that provides a uniform interface to existing neuroimaging software and facilitates interaction between different software components. The *Nipype* project was born in the neuroimaging community, and has been helping scientists build workflows for a decade, providing a uniform interface to such neuroimaging packages as FSL [WJP+09], ANTs [ATS09], AFNI [Cox96], FreeSurfer [DFS99] and SPM [FAK+07]. This flexibility has made it an ideal basis for popular preprocessing tools, such as fMRIPrep [OEG19] and C-PAC [C-P]. The second generation of *Nipype* ecosystem is meant to provide additional flexibility and is being developed with reproducibility, ease of use, and scalability in mind. *Pydra* itself is a standalone project and is designed as a general-purpose dataflow engine to support any scientific domain.

The goal of *Pydra* is to provide a lightweight dataflow engine for computational graph construction, manipulation, and distributed execution, as well as ensuring reproducibility of scientific pipelines. In *Pydra*, a dataflow is represented as a directed acyclic graph, where each node represents a Python function, execution of an external tool, or another reusable dataflow. The combination of several key features makes *Pydra* a customizable and powerful dataflow engine:

- **Composable dataflows:** Any node of a dataflow graph can be another dataflow, allowing for nested dataflows of arbitrary depths and encouraging creating reusable dataflows.
- **Flexible semantics for creating nested loops over input sets:** Any *Task* or dataflow can be run over input parameter sets and the outputs can be recombined (similar concept to Map-Reduce model [DG04], but *Pydra* extends this to graphs with nested dataflows).
- **A content-addressable global cache:** Hash values are computed for each graph and each Task. This supports reusing of previously computed and stored dataflows and *Tasks*.
- **Support for Python functions and external (shell) commands:** *Pydra* can decorate and use existing functions in Python libraries alongside external command line tools, allowing easy integration of existing code and software.
- **Native container execution support:** Any dataflow or *Task* can be executed in an associated container (via Docker or Singularity) enabling greater consistency for reproducibility.
- **Auditing and provenance tracking:** *Pydra* provides a simple JSON-LD -based message passing mechanism to capture the dataflow execution activties as a provenance graph. These messages track inputs and outputs of each task in a dataflow, and the resources consumed by the task.

Pydra is a pure Python 3.7+ package with a limited set of dependencies, which are themselves only dependent on the Python Standard library. It leverages *type annotation* and *AsyncIO* in its core operations. *Pydra* uses the *attr* package for extended annotation and validation of inputs and outputs of tasks, the *cloudpickle* package to pickle interactive task definitions, and the *pytest* testing framework. *Pydra* is intended to help scientific workflows which rely on significant file-based operations and

* *Corresponding author: djarecka@gmail.com*
‡ *Massachusetts Institute of Technology, Cambridge, MA, USA*
¶ *Stanford University, Stanford, CA, USA*
§ *Stony Brook University School of Medicine, Stony Brook, NY, USA*

which evaluate outcomes of complex dataflows over a hyper-space of parameters. It is important to note, that *Pydra* is not a framework for writing efficient scientific algorithms or for use in applications where caching and distributed execution are not necessary. Since *Pydra* relies on a filesystem cache at present, it is also not designed for dataflows that need to operate purely in memory.

The next section will describe the *Pydra* architecture --- main package classes and interactions between them. The *Key Features* section focuses on a set of features whose combination distinguishes *Pydra* from other dataflow engines. The paper concludes with a set of applied examples demonstrating the power and utility of *Pydra*, and short discussion on the future directions.

Architecture

Pydra architecture has three core components: *Task*, *Submitter* and *Worker*. *Tasks* form the basic building blocks of the dataflow, while *Submitter* orchestrates the dataflow execution model. Different types of *Workers* allow *Pydra* to execute the task on different compute architectures. Fig. 1 shows the Class hierarchy and interaction between them in the present *Pydra* architecture. It was designed this way to decouple *Tasks* and *Workers*. In order to describe *Pydra*'s most notable features in the next section, we briefly describe the role of each of these classes.

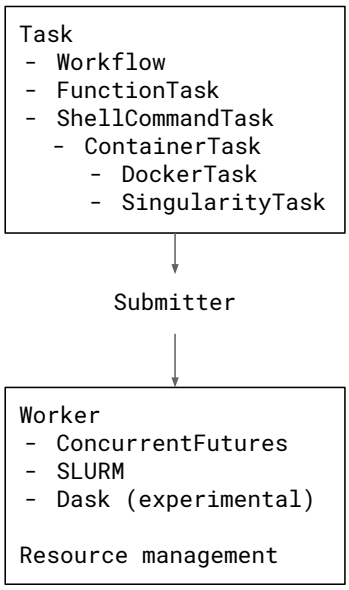

Fig. 1: *A schematic presentation of principal classes in Pydra.*

Dataflows Components: Task and Workflow

A *Task* is the basic runnable component of *Pydra* and is described by the class `TaskBase`. A *Task* has named inputs and outputs, thus allowing construction of dataflows. It can be hashed and executes in a specific working directory. Any *Pydra*'s *Task* can be used as a function in a script, thus allowing dual use in *Pydra*'s *Workflows* and in standalone scripts. There are several classes that inherit from `TaskBase` and each has a different application:

- `FunctionTask` is a *Task* that executes Python functions. Most Python functions declared in an existing library, package, or interactively in a terminal can be converted to a `FunctionTask` by using *Pydra*'s decorator - `mark.task`.

```
import numpy as np
from pydra import mark
fft = mark.annotate({'a': np.ndarray,
                     'return': float})(np.fft.fft)
fft_task = mark.task(fft)()
result = fft_task(a=np.random.rand(512))
```

fft_task is now a *PydraTask* and result will contain a *Pydra*'s `Result` object. In addition, the user can use Python's function annotation or another *Pydra* decorator—`mark.annotate` in order to specify the output. In the following example, we decorate an arbitrary Python function to create named outputs:

```
@mark.task
@mark.annotate(
    {"return": {"mean": float, "std": float}}
)
def mean_dev(my_data):
    import statistics as st
    return st.mean(my_data), st.stdev(my_data)

result = mean_dev(my_data=[...])()
```

When the *Task* is executed *result.output* will contain two attributes: *mean* and *std*. Named attributes facilitate passing different outputs to different downstream nodes in a dataflow.

- `ShellCommandTask` is a *Task* used to run shell commands and executables. It can be used with a simple command without any arguments, or with specific set of arguments and flags, e.g.:

```
ShellCommandTask(executable="pwd")

ShellCommandTask(executable="ls", args="my_dir")
```

The *Task* can accommodate more complex shell commands by allowing the user to customize inputs and outputs of the commands. One can generate an input specification to specify names of inputs, positions in the command, types of the inputs, and other metadata. As a specific example, FSL's BET command (Brain Extraction Tool) can be called on the command line as:

```
bet input_file output_file -m
```

Each of the command argument can be treated as a named input to the `ShellCommandTask`, and can be included in the input specification. As shown next, even an output is specified by constructing the *out_file* field form a template:

```
bet_input_spec = SpecInfo(
    name="Input",
    fields=[
    ( "in_file", File,
      { "help_string": "input file ...",
        "position": 1,
        "mandatory": True } ),
    ( "out_file", str,
      { "help_string": "name of output ...",
        "position": 2,
        "output_file_template":
                          "{in_file}_br" } ),
    ( "mask", bool,
      { "help_string": "create binary mask",
        "argstr": "-m", } ) ],
    bases=(ShellSpec,) )

ShellCommandTask(executable="bet",
                 input_spec=bet_input_spec)
```

Outputs can also be specified separately using a similar output specification.

- `ContainerTask` class is a child class of `ShellCommandTask` and serves as a parent class for `DockerTask` and `SingularityTask`. Both *Container Tasks* run shell commands or executables within containers with specific user defined environments using *Docker* [doc] and *Singularity* [sin] software respectively. This might be extremely useful for users and projects that require environment encapsulation and sharing. Using container technologies helps improve scientific workflows reproducibility, one of the key concept behind *Pydra*.
  These *Container Tasks* can be defined by using `DockerTask` and `SingularityTask` classes directly, or can be created automatically from `ShellCommandTask`, when an optional argument `container_info` is used when creating a *Shell Task*. The following two types of syntax are equivalent:

```
DockerTask(executable="pwd", image="busybox")

ShellCommandTask(executable="ls",
     container_info=("docker", "busybox"))
```

- `Workflow` - is a subclass of *Task* that provides support for creating *Pydra* dataflows. As a subclass, a *Workflow* acts like a *Task* and has inputs, outputs, is hashable, and is treated as a single unit. Unlike *Tasks*, workflows embed a directed acyclic graph. Each node of the graph contains a *Task* of any type, including another *Workflow*, and can be added to the *Workflow* simply by calling the `add` method. The connections between *Tasks* are defined by using so called *Lazy Inputs* or *Lazy Outputs*. These are special attributes that allow assignment of values when a *Workflow* is executed rather than at the point of assignment. The following example creates a *Workflow* from two *Pydra Tasks*.

```
# creating workflow with two input fields
wf = Workflow(input_spec=["x", "y"])
# adding a task and connecting task's input
# to the workflow input
wf.add(mult(name="mlt",
               x=wf.lzin.x, y=wf.lzin.y))
# adding anoter task and connecting
# task's input to the "mult" task's output
wf.add(add2(name="add", x=wf.mlt.lzout.out))
# setting worflow output
wf.set_output([("out", wf.add.lzout.out)])
```

State

All *Tasks*, including *Workflows*, can have an optional attribute representing an instance of the `State` class. This attribute controls the execution of a *Task* over different input parameter sets. This class is at the heart of *Pydra's* powerful *Map-Reduce* over arbitrary inputs of nested dataflows feature. The `State` class formalizes how users can specify arbitrary combinations. Its functionality is used to create and track different combinations of input parameters, and optionally allow limited or complete recombinations. In order to specify how the inputs should be split into parameter sets, and optionally combined after the *Task* execution, the user can set *splitter* and *combiner* attributes of the `State` class. These attributes can be set by calling `split` and `combine` methods in the *Task* class. Here we provide a simple *Map-Reduce* example:

```
task_with_state =
      add2(x=[1, 5]).split("x").combine("x")
```

In this example, the `State` class is responsible for creating a list of two separate inputs, *[{x: 1}, {x:5}]*, each run of the *Task* should get one element from the list. The results are grouped back when returning the result from the *Task*. While this example illustrates mapping and grouping of results over a single parameter, *Pydra* extends this to arbitrary combinations of input fields and downstream grouping over nested dataflows. Details of how splitters and combiners power *Pydra*'s scalable dataflows are described later.

Submitter

The `Submitter` class is responsible for unpacking *Workflows* and single *Tasks* with or without `State` into standalone stateless jobs, *runnables*, that are then executed by *Workers*. When the *runnable* is a *Workflow*, the *Submitter* is responsible for checking if the *Tasks* from the graph are ready to run, i.e. if all the inputs are available, including the inputs that are set to the *Lazy Outputs* from previous *Tasks*. Once a *Task* is ready to run, the *Submitter* sends it to a *Worker*. When the runnable has a *State*, then the *Submitter* unpacks the *State* and sends multiple jobs to the *Worker* for the same *Task*. In order to avoid memory consumption as a result of scaling of *Tasks*, each job is sent as a pointer to a pickle file, together with information about its state, so that proper input can be retrieved just before running the *Task*. *Submitter* uses *AsyncIO* to manage all job executions to work in parallel, allowing scaling of execution as *Worker* resources are made available.

Workers

Workers in *Pydra* are responsible for the actual execution of the *Tasks* and are initialized by the *Submitter*. *Pydra* supports three types of execution managers: *ConcurrentFutures*, *Slurm* and *Dask* (experimental). When `ConcurrentFuturesWorker` is created, `ProcessPoolExecutor` is used to create a "pool" for adding the runnables. `SlurmWorker` creates an'sbatch' submission script in order to execute the task, and `DaskWorker` make use of Dask's `Client` class and its `submit` method. All workers use *async functions* from *AsyncIO* in order to handle asynchronous processes. All *Workers* rely on a `load_and_run` function to execute each job from its pickled state.

Key Features

In this section, features of *Pydra* that exemplify its utility for scientific dataflows are presented. Individually, some of these features are present in the numerous workflow packages that exist, but *Pydra* is the only software that brings them together using a very lighweight codebase. The combination of the following features makes *Pydra* a powerful tool in scientific computation.

Nested and Hashed Workflows

Scientific dataflows typically involve significant refinement and extensions as science and instrumentation evolves. *Pydra* was designed to provide an easy way of creating scientific dataflows that range from simple linear pipelines to complex nested graphs. It enables reproducibility and reduces cost of dataflow maintenance through flexible reuse of already existing functions and *Workflows* in new applications. The `Workflow` class inherits

from `TaskBase` class and can be treated by users as any other *Task*, so can itself be added as a node in another *Workflow*. This provides an easy way of creating nested *Workflows* of arbitrary depth, and reuse already existing *Workflows*. This is schematically shown in Fig. 2.

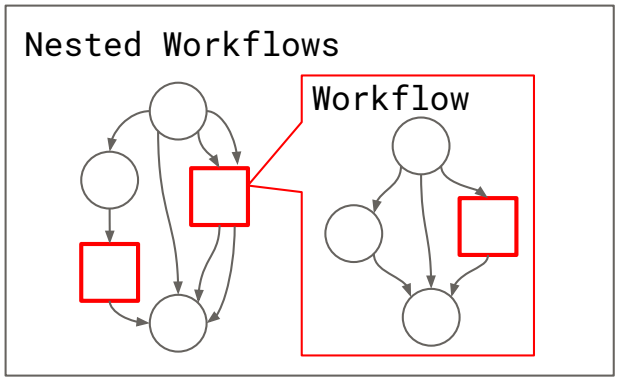

Fig. 2: *A nested Pydra Workflow, black circles represent single Tasks, and Workflows are represented by red rectangles.*

The *Pydra*'s *Submitter* supports this nested architecture and can dynamically extend the execution graph. Since a *Workflow* works like a *Task*—has inputs, outputs, and is hashable, once executed it does not need to recompute its operations if cached (*Pydra*'s caching is explained later in the section).

State and Nested Loops over Input

One of the main goals of creating *Pydra* was to support flexible evaluation of a *Task* or a *Workflow* over combinations of input parameters. This is the key feature that distinguishes it from most other dataflow engines. This is similar to the concept of the *Map-Reduce* [DG04], but extends it to work over arbitrary nested graphs. In complex dataflows, this would typically involve significant overhead for data management and use of multiple nested loops. In *Pydra*, this is controlled by setting specific `State` related attributes through *Task* methods. In order to set input splitting (or mapping), *Pydra* requires setting up a *splitter*. This is done using *Task*'s `split` method. The simplest example would be a *Task* that has one field x in the input, and therefore there is only one way of splitting its input. Assuming that the user provides a list as a value of x, *Pydra* splits the list, so each copy of the *Task* will get one element of the list. This can be represented as follow:

$$S = x : x = [x_1, x_2, ..., x_n] \longmapsto x = x_1, x = x_2, ..., x = x_n \ ,$$

where S represents the *splitter*, and x is the input field.

That is also represented in Fig. 3, where *x=[1, 2, 3]* as an example.

Scalar and outer splitters: Whenever a *Task* has more complicated inputs, i.e. multiple fields, there are two ways of creating the mapping, each one is used for different application. These *splitters* are called *scalar splitter* and *outer splitter* They use a special, but Python-based syntax as described next.

A *scalar splitter* performs element-wise mapping and requires that the lists of values for two or more fields to have the same length. The *scalar splitter* uses Python tuples and its operation is therefore represented by a parenthesis, `()`:

$$\begin{aligned} S = (x,y) \quad &: \quad x = [x_1, x_2, .., x_n], \ \ y = [y_1, y_2, .., y_n] \\ &\mapsto \quad (x,y) = (x_1, y_1), ..., (x,y) = (x_n, y_n), \end{aligned}$$

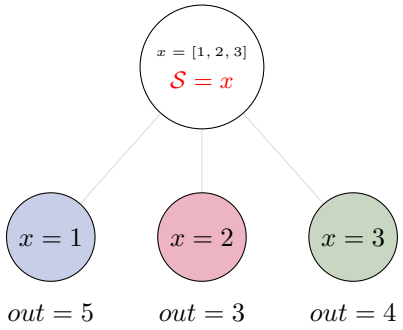

Fig. 3: *Diagram representing a Task with one input and a simple splitter. The white node represents an original Task with x=[1,2,3] as an input and S=x as a splitter. The coloured nodes represent stateless copies of the original Task after splitting the input, these are the runnables that are executed by Workers.*

where S represents the *splitter*, x and y are the input fields.

This is also represented as a diagram in Fig. 4

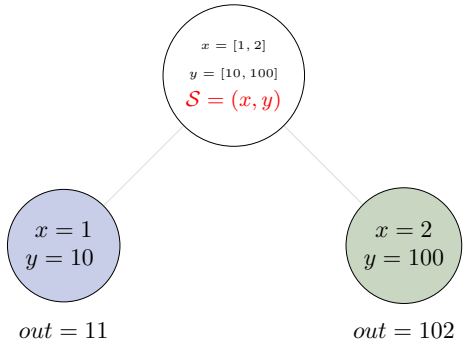

Fig. 4: *Diagram representing a Task with two input fields and a scalar splitter. The symbol convention is described in 3.*

The second option of mapping the input, when there are multiple fields, is provided by the *outer splitter*. The *outer splitter* creates all combination of the input values and does not require the lists to have the same lengths. The *outer splitter* uses Python's list syntax and is represented by square brackets, `[]`:

$$\begin{aligned} S = [x,y] \quad &: \quad x = [x_1, x_2, ..., x_n], \ \ y = [y_1, y_2, ..., y_m], \\ &\mapsto \quad (x,y) = (x_1, y_1), (x,y) = (x_1, y_2) ..., (x,y) = (x_n, y_m). \end{aligned}$$

The *outer splitter* for a node with two input fields is schematically represented in Fig. 5

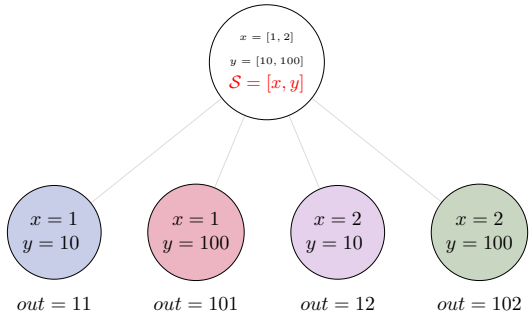

Fig. 5: *Diagram representing a Task with two input fields and an outer splitter. The symbol convention is described in 3.*

Different types of splitters can be combined over inputs such as *[inp1, (inp2, inp3)]*. In this example an *outer splitter* provides

all combinations of values of *inp1* with pairwise combinations of values of *inp2* and *inp3*. This can be extended to arbitrary complexity.

Combiners: In addition to the splitting the input, *Pydra* supports grouping or combining the output resulting from the splits. Taking as an example the simple *Task* represented in Fig. 3, in some application it can be useful to group all output values of the individual splits. In order to achieve this for a *Task*, a user can specify a *combiner*. This can be set by calling `combine` method. Note, the *combiner* only makes sense when a *splitter* is set first. When *combiner=x*, all values are combined together within one list, and each element of the list represents an output of the *Task* for the specific value of the input x. Splitting and combining for this example can be written as follows:

$$\begin{aligned} S = x \quad &: \quad x = [x_1, x_2, ..., x_n] \mapsto x = x_1, x = x_2, ..., x = x_n, \\ C = x \quad &: \quad out(x_1), ..., out(x_n) \mapsto out_{comb} = [out(x_1), ...out(x_n)], \end{aligned}$$

where S represents the *splitter*, C represents the *combiner*, x is the input field, $out(x_i)$ represents the output of the *Task* for x_i, and out_{comb} is the final output after applying the *combiner*.

In the situation where input has multiple fields and an *outer splitter* is used, there are various ways of combining the output. Taking as an example *Task* represented in Fig. 5, user might want to combine all the outputs for one specific value of :math:x_i and all the values of :math:y. In this situation, the combined output would be a two dimensional list, each inner list for each value of :math:x. This is written as follows:

$$\begin{aligned} C = y \quad &: \quad out(x_1, y1), out(x_1, y2), ...out(x_n, y_m) \\ &\longmapsto \quad [[out(x_1, y_1), ..., out(x_1, y_m)], \\ &\qquad ..., \\ &\qquad [out(x_n, y_1), ..., out(x_n, y_m)]]. \end{aligned}$$

And is represented in Fig. 6.

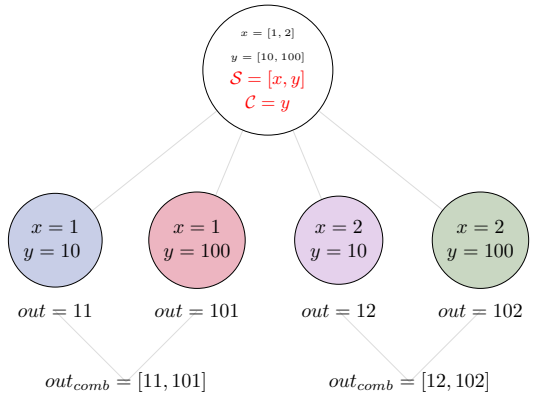

Fig. 6: *Diagram representing a Task with two input fields, an outer splitter and a combiner. The white node represents an original Task with x=[1,2], y=[10, 100] as an input, S=[x, y] as a splitter, and C=y as a combiner. The coloured nodes represent stateless copies of the original Task after splitting the input, these are the runnables that are executed by Workers. At the end outputs for all values of y are combined together within out_{comb}.*

However, for the diagram from 5, the user might want to combine all values of x for specific values of y. One may also need to combine all the values together. This can be achieved by providing a list of fields, *[x, y]* to the combiner. When a full combiner is set, i.e. all the fields from the splitter are also in the combiner, the output is a one dimensional list:

$$C = [x, y] : out(x_1, y1), ...out(x_n, y_m) \longmapsto [out(x_1, y_1), ..., out(x_n, y_m)].$$

And is represented in Fig. 7.

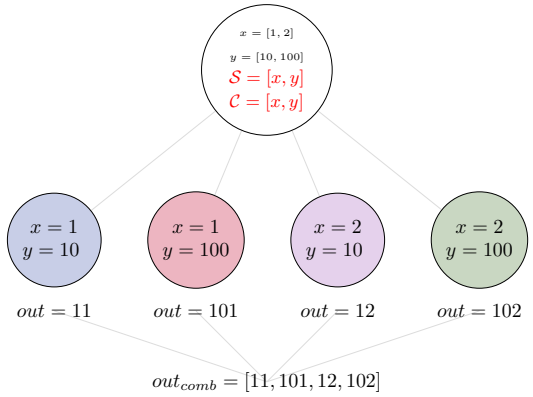

Fig. 7: *Diagram representing a Task with two input fields, an outer splitter and a full combiner. The Tasks are run in exactly the same way as previously, but at the end all of the output values are combined together. The symbol convention is described in 6.*

These are the basic examples of the *Pydra*'s *splitter-combiner* concept. It is important to note, that *Pydra* allows for mixing *splitters* and *combiners* on various levels of a dataflow. They can be set on a single *Task* or a *Workflow*. They can be passed from one *Task* to following *Tasks* within the *Workflow*. Examples of this more complex operation are presented in the next section.

Checksums and Global Cache

One of the key feature of *Pydra* is the support for a *Global Cache*. This allows multiple people in a laboratory, or even across laboratories to use each other's execution outputs on the same data without having to rerun the same computation. Each *Task* and *Workflow* has an attribute called *checksum*. In order to create the *checksum*, all of the input fields are collected and hash value is calculated. If *File* or *Directory* is used as an input, than the hash value of the content is used. For *Workflows*, the connections between the *Tasks* are also included in the final *checksum*, and hence the checksum of a *Workflow* changes if its underlying graph changes. The *checksum* is used to create output directory path during execution and can be reused in future executions of the same exact *Task* or *Workflow*. To reuse, a user can specify `cache_dir` and `cache_locations` when creating a *Task* or *Workflow*. The `cache_dir` is a read-write path, where you want your outputs to be saved, but `cache_location` can include a list of paths, which allow re-using existing caches. Before running any *Task* or *Workflow*, *Pydra* checks all the directories that are either in `cache_dir` or `cache_locations`, and if the specific checksum is found, then the results are reloaded without running the specific *Task*. It is important to emphasize that without a cache, every element of a nested *Workflow* would be re-executed. Using *Global Cache* can significantly reduce execution time when the same operations on the same data are repeated. This is also true for *Tasks* with *State*. If the number of input elements is expanded, the previously cached results can be reused without recomputation. For scientific workflows, where many tasks take significant computational resources, this can drastically speed up reruns.

Applications and Examples

In this section, we highlight *Pydra* through two examples. The first example is an intuitive scientific Python example to demonstrate the power of *Pydra*'s splitter and combiner. The second example extends this demonstration with a more practical machine learning model comparison workflow leveraging scikit-learn.

Example 1: Sine Function Approximation

This example illustrates the flexibility of the *Pydra*'s *splitters* and *combiners*, but the example is not meant to convince scientist to use *Pydra* to write algorithms like this. The exemplary workflow will calculate the approximated values of *Sine* function for various values of *x*. The *Workflow* uses the Taylor polynomial formula for *Sine* function:

$$\sum_{n=0}^{n_{max}} \frac{(-1)^n}{(2n+1)!} x^{2n+1} = x - \frac{x^3}{3!} + \frac{x^5}{5!} + \ldots$$

where n_{max} is a degree of approximation.

Since the idea is to make the execution as embarassingly parallel as possible, each of the term for each value of *x* should be calculated separately. This is done by function *term (x, n)*. In addition, *range_fun(n_max)* is used to return a list of integers from *0* to *n_max* and *summing(terms)* will sum all the terms for the specific value of *x* and *n_max*.

```
from pydra import Workflow, Submitter, mark
import math

@mark.task
def range_fun(n_max):
    return list(range(n_max+1))

@mark.task
def term(x, n):
    import math
    fract = math.factorial(2 * n + 1)
    polyn = x ** (2 * n + 1)
    return (-1)**n * polyn / fract

@mark.task
def summing(terms):
    return sum(terms)
```

The *Workflow* takes two inputs - a list of values of *x* and a list of values of *n_max*. In order to calculate various degrees of the approximation for each value of *x*, an *outer splitter* is used *[x, n_max]*. All approximations for a specific values of *x* is aggregated by using *n_max* as a combiner.

```
wf = Workflow(name="wf", input_spec=["x", "n_max"])
wf.split(["x", "n_max"]).combine("n_max")
wf.inputs.x = [0, 0.5 * math.pi, math.pi]
wf.inputs.n_max = [2, 4, 10]
```

All three *Function Tasks* are added to the *Workflow* and connected together using *lazy* connections. The second task, *term*, has to be additionally split over *n* to compute the different pieces of the Taylor approximation and the results of each term calculation are grouped together through the *combine* method.

```
wf.add(range_fun(name="range", n_max=wf.lzin.n_max))
wf.add(term(name="term", x=wf.lzin.x,
                         n=wf.range.lzout.out).
       split("n").combine("n"))
wf.add(summing(name="sum", terms=wf.term.lzout.out))
```

Finally, the *Workflow* output is set as the approximation using `set_output` method. Thus the *Workflow* reflects a parallelizable self contained function.

```
wf.set_output([("sin", wf.sum.lzout.out)])
res = wf(plugin="cf")
```

When executed using the concurrent futures library, the result is a two dimensional list of *Results*. For each value of *x* the *Workflow* computes a list of three approximations. As an example, for *x=pi/2* this returns the following list:

```
[...[Result(output=Output(sin=1.0045248555348174),
            runtime=None, errored=False),
     Result(output=Output(sin=1.0000035425842861),
            runtime=None, errored=False),
     Result(output=Output(sin=1.0000000000000002),
            runtime=None, errored=False)],
...]
```

Each *Result* contains three elements: *output* reflecting the actual computed output, *runtime* reflecting the information related to resources used during execution (when a resource audit flag is set), and *errored* a boolean flag which indicates whether the task errored or not. As expected, the values of the *Sine* function are getting closer to *1* with increasing degree of the approximation.

The described *Workflow* is schematically presented in Fig. 8.

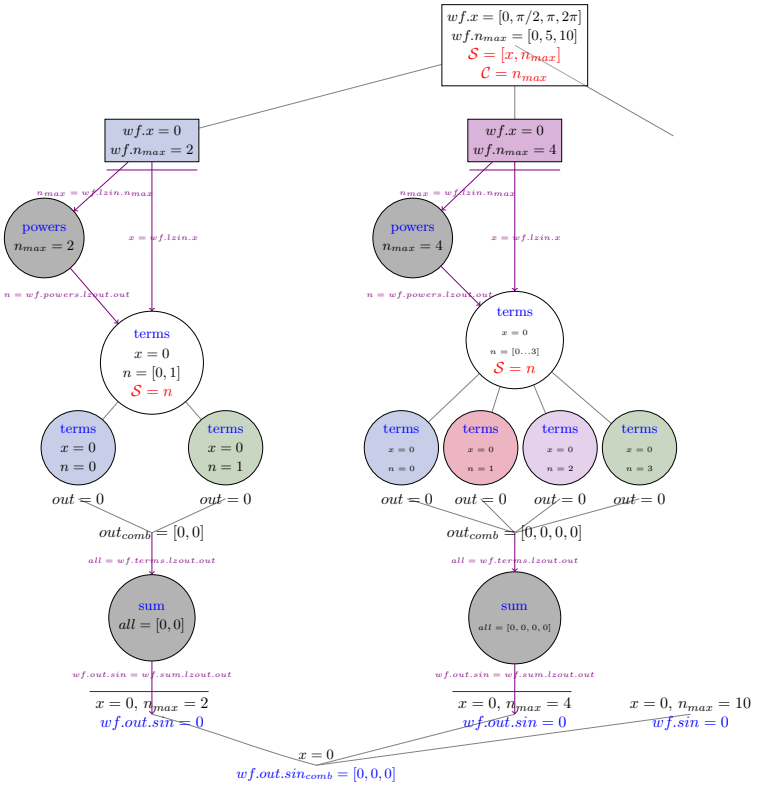

Fig. 8: *Diagram representing part of the Workflow for calculating* Sine *function approximations of various degrees for values of x. Circles represent single Tasks and rectangles represent Workflows. The white nodes represent Task or Workflow with a State. The coloured nodes represent stateless copies of the original Task after splitting the input. The gray nodes represent a Task that has no State.*

Example2: Machine Learning Model Comparison

The massive parameter search space of models and their parameters makes machine learning an ideal use case for *Pydra*. This section illustrates a general-purpose machine learning *Pydra*'s *Workflow* for model comparison using a boostrapped shuffle-split mechanism for choosing training and test pairs from a given dataset. The example leverages *Pydra*'s powerful splitters and

combiners to scale across a set of classifiers and metrics. It also uses *Pydra*'s caching to not redo model training and evaluation when new metrics are added, or when number of iterations is increased. The complete model comparison workflow is available as an installable package called *pydra-ml* [pyd], and includes SHAP-based feature importance evaluation in addition to model comparison.

The *Workflow* presented here comprises four *FunctionTasks*. For the sake of clarity, we will not redisplay the task code here. They can be found in the *tasks.py* file in *pydra-ml* [pyd]. The first function, *read_data*, reads csv data as a *pandas.DataFrame* and allows the user to extract specific columns as the input, *X*, to a learning model, a target column, *y*, and an optional *group* column. The second function, *gen_splits*, uses *GroupShuffleSplit* from *sklearn.model_selection* to generate a set of train-test splits given *n_splits* and *test_size*, with an option to define *group* and *random_state*. It returns *train_test_splits* and *split_indices*. The main function to train the classifier, *train_test_kernel*, takes as input a specific train-test split pair, a target variable, a parameter providing information about which classifier to use and whether to generate a null model by permuting the labels. The final function *calc_metric* returns the value from a scoring function given the actual target and predicted values from the classifier.

These tasks are combined together within a *Workflow* exploiting splitters and combiners. The *Workflow* itself has an *outer split* for *clf_info* and *permute*, allowing evaluation of null and non-null models for every classifier. The core model fitting and evaluation function *train_test_kernel* uses an internal splitter to iterate over all the bootstrapped iterations. Using *Pydra*, it is possible to split over *split_index*, that comes from *gensplit Task*, and run *train_test_kernel* for each of them without combining. This maintains `State` which can be used by the *calc_metric* function to evaluate different scoring methods on the classifier outputs and combine these results back together.

```
wf = pydra.Workflow(name="ml_wf",
          input_spec=list(inputs.keys()),
          **inputs,
          cache_dir=cache_dir,
          cache_locations=cache_locations)
# Workflow level splitting over combination
# of values
wf.split(["clf_info", "permute"])
wf.add(read_file(
        name="readcsv",
        filename=wf.lzin.filename,
        x_indices=wf.lzin.x_indices,
        target_vars=wf.lzin.target_vars))
wf.add(gen_splits(
        name="gensplit",
        n_splits=wf.lzin.n_splits,
        test_size=wf.lzin.test_size,
        X=wf.readcsv.lzout.X,
        Y=wf.readcsv.lzout.Y,
        groups=wf.readcsv.lzout.groups))
wf.add(train_test_kernel(
        name="fit_clf",
        X=wf.readcsv.lzout.X,
        y=wf.readcsv.lzout.Y,
        train_test_split=wf.gensplit.lzout.splits,
        split_index=wf.gensplit.lzout.split_indices,
        clf_info=wf.lzin.clf_info,
        permute=wf.lzin.permute))
# Task level splitting over bootstrapped
# train-test pairs
wf.fit_clf.split("split_index")
wf.add(calc_metric(
        name="metric",
        output=wf.fit_clf.lzout.output,
        metrics=wf.lzin.metrics))
# Downstream combination after calculating
# a set of metrics on each train-test pair
wf.metric.combine("fit_clf.split_index")
wf.set_output(
    [
     ("output", wf.metric.lzout.output),
     ("score", wf.metric.lzout.score),
     ("feature_names",
            wf.readcsv.lzout.feature_names),
    ]
)
```

The workflow is executed by providing an input dictionary exemplary input dictionary and the *Workflow*'s submission can look as follow:

```
clfs = [
 ('sklearn.ensemble', 'ExtraTreesClassifier',
  dict(n_estimators=100)),
 ('sklearn.neural_network', 'MLPClassifier',
  dict(alpha=1, max_iter=1000)),
 ('sklearn.neighbors', 'KNeighborsClassifier', dict(),
 [{'n_neighbors': [3, 7, 15],
   'weights': ['uniform','distance']}]),
 ('sklearn.ensemble', 'AdaBoostClassifier', dict())]

inputs = {"filename": 'iris.csv',
          "x_indices": range(4), "target_vars": ("label"),
          "n_splits": 3, "test_size": 0.2,
          "metrics": ["roc_auc_score"],
          "permute": [True, False], "clf_info": clfs}
n_procs = 8 # for parallel processing
cache_dir = os.path.join(os.getcwd(), 'cache')
wf_cache_dir = os.path.join(os.getcwd(), 'cache-wf')

# Execute the workflow in parallel using multiple processes
with pydra.Submitter(plugin="cf", n_procs=n_procs) as sub:
    sub(runnable=wf)

result = wf.result(return_inputs=True)
```

The result from the *Workflow* is a set of scores for permuted and non-permuted models. This is a list, each element of the list is for one value of *clf_info* and *permute*, both fields were set as input fields to the *Workflow*. All *Result* objects have an *output.score* field that is also a list. Each element of the *score* corresponds to a different value of *split_index*, that was set both as a *splitter* and *combiner* to the *fit_cls Task*. This gives an option to easily compare various models and sets of parameters.

```
[({'ml_wf.clf_info':
      ('sklearn.ensemble','ExtraTreesClassifier',
       {'n_estimators': 100}),
  'ml_wf.permute': True},
 Result(output=Output(score=[0.2622, 0.1733, 0.2975]),
        runtime=None, errored=False)),
({'ml_wf.clf_info':
       ('sklearn.ensemble', 'ExtraTreesClassifier',
        {'n_estimators': 100}),
  'ml_wf.permute': False},
 Result(output=Output(score=[1.0, 0.9333, 0.9333]),
        runtime=None, errored=False)),

...

({'ml_wf.clf_info':
      ('sklearn.ensemble', 'AdaBoostClassifier', {}),
  'ml_wf.permute': False},
 Result(output=Output(score=[0.9658, 0.9333, 0.8992]),
        runtime=None, errored=False))]
```

Usually, there is no easy way in *scikit-learn* to compare models in parallel across a variety of classifiers without using loops. It is possible to do all this natively in scikit-learn and joblib, but would

require much more code to do the maintenance of the dataflow and aggregation.

Summary and Future Directions

Pydra is a new lightweight dataflow engine written in Python. The combination of several key features - including flexible option for splitting and combining input fields, and *Global Cache* - makes *Pydra* a customizable and powerful dataflow engine. The *Pydra*'s developers are mostly from the Neuroimaging community, which provides a plethora of use-cases for complex dataflows, but the package is designed as a general-purpose dataflow engine to support any scientific domain. As the next step, the developer team would like to invite more scientist to use *Pydra* in order to test the package for diverse applications. In the near future, the developer team is also planning to work on:

- improvement of *Worker* classes to coordinate resource management
- improved interaction with *Dask* and other resource managers (e.g., SLURM) in HPC and Cloud environments.
- updates to the *Nipype* software to use *Pydra* as its engine
- improve the documentation and tutorials

We welcome scientists and developers to join the project. The project repository is available on GitHub under *Nipype* organization: https://github.com/nipype/pydra. In addition, there is also a repository that contains Jupyter Notebooks with *Pydra* tutorial: https://github.com/nipype/pydra-tutorial. The tutorial can be run locally or using the Binder service.

Acknowledgements

This was supported by NIH grants P41EB019936, R01EB020740. We thank the neuroimaging community for feedback during development, and Anna Jaruga for her feedback on the paper.

REFERENCES

[ATS09] Brian B Avants, Nick Tustison, and Gang Song. Advanced normalization tools (ants). *Insight j*, 2(365):1–35, 2009.

[C-P] C-PAC. http://fcp-indi.github.io/.

[Cox96] Robert W. Cox. Afni: Software for analysis and visualization of functional magnetic resonance neuroimages. *Computers and Biomedical Research*, 29(3):162 – 173, 1996. URL: http://www.sciencedirect.com/science/article/pii/S0010480996900142, doi:https://doi.org/10.1006/cbmr.1996.0014.

[Dev] Nipype Developers.

[DFS99] Anders M. Dale, Bruce Fischl, and Martin I. Sereno. Cortical surface-based analysis: I. segmentation and surface reconstruction. *NeuroImage*, 9(2):179 – 194, 1999. URL: http://www.sciencedirect.com/science/article/pii/S1053811998903950, doi:https://doi.org/10.1006/nimg.1998.0395.

[DG04] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI'04: Sixth Symposium on Operating System Design and Implementation*, pages 137–150, San Francisco, CA, 2004.

[doc] Docker. https://www.docker.com/.

[FAK+07] K.J. Friston, J. Ashburner, S.J. Kiebel, T.E. Nichols, and W.D. Penny, editors. *Statistical Parametric Mapping: The Analysis of Functional Brain Images*. Academic Press, 2007. URL: http://store.elsevier.com/product.jsp?isbn=9780123725608.

[GBM+11] Krzysztof Gorgolewski, Christopher Burns, Cindee Madison, Dav Clark, Yaroslav Halchenko, Michael Waskom, and Satrajit Ghosh. Nipype: A flexible, lightweight and extensible neuroimaging data processing framework in python. *Frontiers in Neuroinformatics*, 5:13, 2011. URL: https://www.frontiersin.org/article/10.3389/fninf.2011.00013, doi:10.3389/fninf.2011.00013.

[OEG19] Ross W. Blair Craig A. Moodie A. Ilkay Isik Asier Erramuzpe James D. Kent Mathias Goncalves Elizabeth DuPre Madeleine Snyder Hiroyuki Oya Satrajit S. Ghosh Jessey Wright Joke Durnez Russell A. Poldrack Oscar Esteban, Christopher J. Markiewicz and Krzysztof J. Gorgolewski. fmriprep: a robust preprocessing pipeline for functional mri. *Nature Methods*, 16:111 – 116, 2019. doi:doi:10.1038/s41592-018-0235-4.

[pyd] pydra-ml. https://github.com/nipype/pydra-ml.

[sin] Singularity. https://sylabs.io/docs/.

[WJP+09] Mark W. Woolrich, Saad Jbabdi, Brian Patenaude, Michael Chappell, Salima Makni, Timothy Behrens, Christian Beckmann, Mark Jenkinson, and Stephen M. Smith. Bayesian analysis of neuroimaging data in fsl. *NeuroImage*, 45(1, Supplement 1):S173 – S186, 2009. Mathematics in Brain Imaging. URL: http://www.sciencedirect.com/science/article/pii/S1053811908012044, doi:https://doi.org/10.1016/j.neuroimage.2008.10.055.