Parsl Documentation

Release 0.1

Yadu Nand Babuji

Nov 18, 2017

Quickstart

Parsl is a Python-based parallel scripting library that supports development and execution of asynchronous and parallel data-oriented workflows (dataflows) that glue together existing executables (called Apps) and functions. Building upon the Swift parallel scripting language, Parsl brings implicit parallel execution support to standard Python scripts. Rather than explicitly defining a graph and/or modifying data structures, instead developers simply annotate selected existing Python functions and Apps. Parsl constructs a dynamic, parallel execution graph derived from the implicit linkage between Apps based on shared input/output data objects. Parsl is resource-independent, that is, the same Parsl script can be executed on arbitrary computational resources from multicore processors through to clusters, clouds, and supercomputers.

CHAPTER 1

Quickstart

To try the parsl tutorial online go here

Getting the latest Parsl is easy, and requires just a few steps:

1.1 Installing

Parsl is now available on PyPI, but first make sure you have Python3.5+

```
>>> python3 --version
```

1.1.1 Installing on Linux

1. Install Parsl:

   ```
   $ python3 -m pip install parsl
   ```

2. Install Jupyter for Tutorial notebooks:

   ```
   $ python3 -m pip install jupyter
   ```

Note: For more detailed info on setting up Jupyter with Python3.5 go here

1.1.2 Installing on Mac OS

1. Install Conda and setup python3.6 following instructions here:

```
$ conda create --name parsl_py36 python=3.6
$ source activate parsl_py36
```

2. Install Parsl:

```
$ python3 -m pip install parsl
```

1.2 For Developers

1. Download Parsl:

```
$ git clone https://github.com/Parsl/parsl.git
```

2. Install:

```
$ cd parsl
$ python3 setup.py install
```

3. Use Parsl!

CHAPTER 2

Requirements

Parsl requires the following :

- Python 3.5+

For testing:

- nose
- coverage

CHAPTER 3

Parsl Tutorial

Parsl allows you to write functions that execute in parallel and tie them together with dependencies to create workflows in python. Parsl wraps python functions into Apps with the **@App** decorator. Decorated function can run in parallel when all their inputs are ready.

For a deeper dive into examples and documentation, please refer our documentation here

```
# Import Parsl
import parsl
from parsl import *
```

Parsl's DataFlowKernel acts as a layer over any pool of execution resources, in our case a pool of [threads](https://en.wikipedia.org/wiki/Thread_(computing)).

```
# Let's create a pool of threads to execute our functions
workers = ThreadPoolExecutor(max_workers=4)
# We pass the workers to the DataFlowKernel which will execute our Apps over the␣
↪workers.
dfk = DataFlowKernel(executors=[workers])
```

3.1 Hello World App

Let's define a simple python function that returns the string 'Hello World!'. This function is made an App using the **@App** decorator. The decorator itself takes the type of app ('python'|'bash') and the DataFlowKernel object as arguments.

```
# Here we define our first App function, a simple python app that returns a string
@App('python', dfk)
def hello ():
    return 'Hello World!'

app_future = hello()
```

3.2 Futures

Unlike a regular python function, when an App is called it returns an AppFuture. Futures act as a proxy to the results or exceptions that the App will produce once its execution completes. You can ask a future object its status with future.done() or ask it to wait for its result with the result() call. It is important to note that while the done() call just gives you the current status, the result() call blocks execution till the App is complete and the result is available.

```
# Check status
print("Status: ", app_future.done())

# Get result
print("Result: ", app_future.result())
```

3.3 Data Dependencies

When a future created by an App is passed as inputs to another, a data dependency is created. Parsl ensures that Apps are executed as their dependencies are resolved.

Let's see an example of this using the monte-carlo method to calculate pi. We call 3 iterations of this slow function, and take the average. The dependency chain looks like this :

```
App Calls     pi()  pi()   pi()
               \     |     /
Futures         a    b    c
                 \   |   /
App Call          mysum()
                     |
Future              avg_pi
```

```
@App('python', dfk)
def pi(total):
    import random       # App functions have to import modules they will use.
    width = 10000       # Set the size of the box in which we drop random points
    center = width/2
    c2  = center**2
    count = 0
    for i in range(total):
        # Drop a random point in the box.
        x,y = random.randint(1, width),random.randint(1, width)
        # Count points within the circle
        if (x-center)**2 + (y-center)**2 < c2:
            count += 1
    return (count*4/total)

@App('python', dfk)
def mysum(a,b,c):
    return (a+b+c)/3
```

3.4 Parallelism

Here we call the function **pi()** three times, each of which run independently in parallel. We then call the next app **mysum()** with the three app futures that were returned from the **pi()** calls. Since **mysum()** is also a parsl app, it returns

an app future immediately, but defers execution (blocks) until all the futures passed to it as inputs have resolved.

```
a, b, c = pi(10**6), pi(10**6), pi(10**6)
avg_pi  = mysum(a, b, c)
```

```
# Print the results
print("A: {0:5} B: {1:5} B: {2:5}".format(a.result(), b.result(), c.result()))
print("Average: {0:5}".format(avg_pi.result()))
```

3.5 Bash Apps

Science aplications often use external software that are invoked from the command line. For instance parameter sweeps with molecular dynamics software such as LAMMPS are very common. Next we will see a simple mocked up science workflow composed of bash apps.

In a bash app function, there are a few special reserved keyword arguments:

- inputs (List) : A list of strings or DataFutures
- outputs (List) : A list of output file paths
- stdout (str) : redirects STDOUT to string filename
- stderr (str) : redirects STDERR to string filename

In addition if a list of output filenames are provided via the outputs=[], a list of DataFutures corresponding to each filename in the outputs list is returned in addition to the AppFuture.

```
@App('bash', dfk)
def sim_mol_dyn(i, dur, outputs=[], stdout=None, stderr=None):
    # The bash app function composes a commandline invocations as a string of␣
↪arbitrary length
    # that is returned by the function. Positional and Keyword args to the fn() are␣
↪formatted
    # into the returned string
    return '''echo "{0}" > {outputs[0]}
    sleep {1};
    ls ;
    '''
# We call sim_mol_dyn with
sim_fut, data_futs = sim_mol_dyn(5, 3, outputs=['sim.out'], stdout='stdout.txt',␣
↪stderr='stderr.txt')
```

```
print(sim_fut, data_futs)
```

CHAPTER 4

Overview

Parsl is designed to enable the composition of asynchronous tasks into workflows in python. Parsl workflows are portable across a variety of computation platforms and exploit many-task parallelism. A workflow is composed in two steps:

1. The markup of functions as parallel functions or `Apps`.
2. Specification of data dependencies between functions.

In Parsl, the execution of an `App` yields futures. These futures can be passed to other `Apps` as inputs, establishing a data-dependency. This allows you to create implicit directed acyclic graphs, though these are never explicitly expressed, either by the programmer or internally in Parsl. `Apps` that have all their dependencies resolved are slated for execution in parallel. This allows Parsl to exploit all parallelism to fullest extent at the granularity expressed by the user.

A MapReduce job can be as simple as this:

```
# Map Function that returns doubles the input integer
@App('python', dfk)
def app_double(x):
    return x*2

# Reduce function that returns the sum of a list
@App('python', dfk)
def app_sum(inputs=[]):
    return sum(inputs)

# Create a list of integers
items = range(0,N)

# Map Phase : Apply an *app* function to each item in list
mapped_results = []
for i in items:
    x = app_double(i)
    mapped_results.append(x)

total = app_sum(inputs=mapped_results)
```

```
print(total.result())
```

CHAPTER 5

Apps

An app is a piece of code that executes asynchronously on an execution resource. An execution resource in this context can be a pool of threads, processes, or even remote workers.

Parsl allows you to markup existing python functions or even snippets of bash script as Apps using the `@App` decorator. We currently support pure python functions by specifying the app type as `python` and calls to external application through bash scripts using app type `bash`.

5.1 Python Apps

The following code snippet shows a simple python function `double(Int)` that has been converted to an App using the `@App` decorator. Note that the first argument to `@App` specifies the App type as python. It is important to note that decorated functions should be pure functions that only act on the input args, and must also explicitly import any modules used.

```
from parsl import *
workers = ThreadPoolExecutor(max_workers=4)
dfk = DataFlowKernel(executors=[workers])

@App('python', data_flow_kernel)
def double(x):
      return x*2

double(x)
```

5.1.1 Limitations

There are limitations on what functions could be converted to apps:

1. Functions should only act only on the inputs
2. Functions should not rely on side-effects such as global variables

3. Parsl uses **'cloudpickle<https://github.com/cloudpipe/cloudpickle>'_** and pickle to serialize Python constructs, such as inputs and outputs to functions. Therefore, Functions can only use inputs and outputs that can be serialized by cloudpickle or pickle.

5.1.2 Special Keywords

Any python function decorated with the `@App` decorator can take a few special reserved keyword arguments.

1. inputs : (list) This keyword argument allows you to pass a list of *Futures*, and thus wait on the results from a list of `Apps`.
2. walltime :(int) This keyword argument is used to specify the duration in seconds for which the function is allowed to run. This keyword will be used in the future, but is not yet enabled.

5.2 Bash Apps

The Bash app allows you to compose calls to external applications from the command-line as you would in a Bash shell. This is made possible by defining a python function that returns the command-line string that is to be executed.

The following code snippet demonstrates a simple bash script written as a string in Python and wrapped as an App. Any command-line invocation represented by an arbitrarily large string, can be returned by a function decorated within an `@App` of type *bash* to be executed. Since most unix tools use files as input and outputs, the decorated *bash* function supports a few special keyword arguments to support files and other needs.

```
@App('bash', data_flow_kernel)
def echo_hello(stderr='std.err', stdout='std.out'):
    return 'echo "Hello World!"'

# echo_hello() when called will execute the string it returns, creating an std.out␣
↪file with
# the contents "Hello World!"
echo_hello()
```

As shown above, special keyword arguments `stdout` and `stderr` passed to a bash app function allow for the capture of the STDOUT and STDERR streams to specific files. The set of special keywords that maybe used are listed below :

5.2.1 Special Keywords

1. inputs : (list) This keyword argument, just like in python apps is used to pass a list of *Futures*, and thus wait on the results from a list of `Apps`.
2. outputs : (list) List of filenames that will be created by the app. This is required so parsl can check if they were created correctly, track them and even move them when being executed on remote machines.
3. stdout : (string) Specify the filepath to a file to which STDOUT should be redirected.
4. stderr : (string) Specify the filepath to a file to which STDERR should be redirected.
5. walltime :(int) This keyword argument is used to specify the duration in seconds for which the function is allowed to run. An AppTimeout exception is raised if walltime is exceeded.

The Bash app allows a user to compose the string to execute on the command-line from the various arguments passed to the decorated function. The string that is returned is formatted by the python string format (PEP 3101).

```
@App('bash', thread_pool_executor)
def echo(arg1, inputs=[], stderr='std.err', stdout='std.out'):
    return 'echo {0} {inputs[0]} {inputs[1]}'

# This call echoes "Hello World !" to the file *std.out*
echo("Hello", inputs=["World", "!"])
```

CHAPTER 6

Futures

When a python function is invoked, the python interpreter waits for the function to complete execution and returns the results. In case of long running functions we may not want to wait for completion and may want the function to be asynchronous. So, in lieu of the results, we return a future. A future is essentially a token that allows us to track the status of an asynchronous task so that we may check the status, results, exceptions, etc. A future is a proxy for a result that is not yet available.

In Parsl, we have two types of futures: AppFutures and DataFutures.

6.1 AppFutures

AppFutures are inherited from the python's concurrent library. An AppFuture is returned by a call to any function that is decorated with Parsl's `@App` decorator. There are four key functionalities that an AppFuture offers us :

1. An AppFuture allows us to check on the current status of launched parsl app, without waiting for it to complete.

```
@App('python', data_flow_kernel)
def double(x):
      return x*2

# doubled_x is an AppFuture
doubled_x = double(10)

# Check status of doubled_x, this will print True if the result is available,␣
↪else false
print(doubled_x.done())
```

2. The AppFuture allows us to block and wait for the result of the launched app:

```
@App('python', data_flow_kernel)
def sleep_double(x):
     import time
     time.sleep(2)    # Sleep for 2 seconds
     return x*2
```

```
# doubled_x is an AppFuture
doubled_x = sleep_double(10)

# The result() waits till the sleep_double() app is done (2s wait) and then prints
# the result from the app *10*
print(doubled_x.result())
```

3. The AppFuture itself can be passed to another decorated app. Such and app will wait untill all of the AppFutures that are inputs are resolved and then proceed with execution.

```
@App('python', data_flow_kernel)
def wait_sleep_double(x, fu_1, fu_2):
     import time
     time.sleep(2)    # Sleep for 2 seconds
     return x*2

# Launch two apps, which will execute in parallel, since they don't have to
# wait on any futures
doubled_x = wait_sleep_double(10, None, None)
doubled_y = wait_sleep_double(10, None, None)

# The third depends on the first two :
#    doubled_x   doubled_y     (2 s)
#           \     /
#           doublex_z          (2 s)
doubled_z = wait_sleep_double(10, doubled_x, doubled_y)

# doubled_z will be done in ~4s
print(doubled_z.result())
```

4. The AppFuture provides a safe way to handle exceptions and errors while executing workflows that are deep and have a range of parallel processing apps.

```
@App('python', data_flow_kernel)
def bad_divide(x):
    return 6/x

# Call bad divide with 0, to cause a divide by zero exception
doubled_x = bad_divide(0)

# Here we can catch and handle the exception.
try:
     doubled_x.result()
except ZeroDivisionError as e:
     print("Oops! You tried to divide by 0 ")
except Exception ase:
     print("Oops! Something really bad happened")
```

In addition to being able to capture the exceptions raised in the specific apps executions represented by AppFutures, Parsl also raises DependencyErrors when apps are unable to execute due to failures in their dependent apps.

6.2 DataFutures

Similar to AppFutures, DataFuture are inherited from the python's concurrent library. While AppFutures represent an asynchronous app task, the DataFuture represents the files it produces. With Bash applications, data flows from one app to another via files. Therefore Parsl needs to keep track of the files produced by an app. This is done by specifying the filenames of outputs as a keyword argument to apps. A list of DataFutures each of which corresponds to the filenames in the `outputs` keyword args, is available through the `outputs` attribute of the AppFuture.

Here's an example :

```
# This app echoes the string passed to it to the first file specified in the
# outputs list
@App('bash', data_flow_kernel)
def echo(message, outputs=[]):
    return 'echo {0} &> {outputs[0]}'

# This app *cat*s the contents of the first file in its inputs[] kwargs to
# the first file in its outputs[] kwargs
@App('bash', data_flow_kernel)
def cat(inputs=[], outputs=[]):
    return 'cat {inputs[0]} > {outputs[0]}'

#Call echo specifying the outputfile
hello = echo("Hello World!", outputs=['hello1.txt'])

# the outputs attribute of the AppFuture is a list of DataFutures
print(hello.outputs)

#This step *cat*s hello1.txt to hello2.txt
hello2 = cat(inputs=[hello.outputs[0]], outputs=['hello2.txt'])

# Wait for the cat app to complete before trying to read the output file
hello2.result()

with open(hello2.outputs[0].result(), 'r') as f:
     print(f.read())
```

CHAPTER 7

Configuring

Parsl allows the user to compose a workflow that is completely separate from the details of it's execution. So far we've seen how apps can be constructed from pure python as well calls to external applications. Once a workflow is created, the execution substrate on which it is to be executed over needs to described to parsl. There are two ways to do this:

1. **Executors** which use threads, ipyparallel workers etc could be constructed manually

```
from parsl import *
workers = ThreadPoolExecutor(max_workers=4)
dfk = DataFlowKernel(executors=[workers])
```

2. A **config** could be passed to the data flow kernel which will initialize the required resources.

```
from parsl import *
config = {
    "sites" : [
        { "site" : "Local_Threads",
          "auth" : { "channel" : None },
          "execution" : {
              "executor" : "threads",
              "provider" : None,
              "max_workers" : 4
          }
        }],
    "globals" : {"lazyErrors" : True}
}
dfk = DataFlowKernel(config=config)
```

CHAPTER 8

Changelog

8.1 Parsl 0.3.0

Here are the major changes that are included in the Parsl 0.3.0 release.

8.1.1 New functionality

- Arguments to DFK has changed

```
# Old
dfk(executor_obj)

# New, pass a list of executors
dfk(executors=[list_of_executors])

# Alternatively, pass the config from which the DFK will
#instantiate resources
dfk(config=config_dict)
```

- Execution providers have been restructured to a separate repo: libsubmit
- Bash app styles have changes to return the commandline string rather than be assigned to the special keyword *cmd_line*. Please refer to RFC #37 for more details. This is a **non-backward** compatible change.
- Output files from apps are now made available as an attribute of the AppFuture. Please refer #26 for more details. This is a **non-backward** compatible change

```
# This is the pre 0.3.0 style
app_fu, [file1, file2] = make_files(x, y, outputs=['f1.txt', 'f2.txt'])

#This is the style that will be followed going forward.
app_fu = make_files(x, y, outputs=['f1.txt', 'f2.txt'])
[file1, file2] = app_fu.outputs
```

- DFK init now supports auto-start of IPP controllers
- Support for channels via libsubmit. Channels enable execution of commands from execution providers either locally, or remotely via ssh.
- Bash apps now support timeouts.
- Support for cobalt execution provider.

8.1.2 Bug fixes

- Futures have inconsistent behavior in bash app fn body #35
- Parsl dflow structure missing dependency information #30

8.2 Parsl 0.2.0

Here are the major changes that are included in the Parsl 0.2.0 release.

8.2.1 New functionality

- Support for execution via IPythonParallel executor enabling distributed execution.
- Generic executors

8.3 Parsl 0.1.0

Here are the major changes that are included in the Parsl 0.1.0 release.

8.3.1 New functionality

- Support for Bash and Python apps
- Support for chaining of apps via futures handled by the DataFlowKernel.
- Support for execution over threads.
- Arbitrary DAGs can be constructed and executed asynchronously.

8.3.2 Bug Fixes

- Initial release, no listed bugs.

CHAPTER 9

Design and Rationale

9.1 Swift vs Parsl

The following text is not well structured, and is mostly a brain dump that needs to be organized. Moving from Swift to an established language (python) came with its own tradeoffs. We get the backing of a rich and very well known language to handle the language aspects as well as the libraries. However, we lose the parallel evaluation of every statement in a script. The thesis is that what we lose is minimal and will not affect 95% of our workflows. This is not yet substanciated.

Please note that there are two Swift languages: Swift/K and Swift/T . These have diverged in syntax and behavior. Swift/K is designed for grids and clusters runs the java based Karajan (hence, /K) execution framework. Swift/T is a completely new implementation of Swift/K for high-performance computing. Swift/T uses Turbine(hence, /T) and and ADLB runtime libraries for highly scalable dataflow processing over MPI, without single-node bottlenecks.

9.1.1 Parallel Evaluation

In Swift (K&T), every statement is evaluated in parallel.

```
y = f(x);
z = g(x);
```

We see that y and z are assigned values in different order when we run Swift multiple times. Swift evaluates both statements in parallel and the order in which they complete is mostly random.

We will *not* have this behavior in Python. Each statement is evaluated in order.

```
int[] array;
foreach v,i in [1:5] {
   array[i] = 2*v;
}

foreach v in array {
```

```
   trace(v)
}
```

Another consequence is that in Swift, a foreach loop that consumes results in an array need not wait for the foreach loop that fill the array. In the above example, the second foreach loop makes progress along with the first foreach loop as it fills the array.

In parsl, a for loop that **launches** tasks has to complete launches before the control may proceed to the next statement. The first for loop has to simply finish iterating, and launching jobs, which should take ~length_of_iterable/1000 (items/task_launch_rate).

```
futures = {};

for i in range(0,10):
    futures[i] = app_double(i);

for i in fut_array:
    print(i, futures[i])
```

The first for loop first fills the futures dict before control can proceed to the second for loop that consumes the contents.

The main conclusion here is that, if the iteration space is sufficiently large (or the app launches are throttled), then it is possible that tasks that are further down the control flow have to wait regardless of their dependencies being resolved.

9.1.2 Mappers

In Swift/K, a mapper is a mechanism to map files to variables. Swift need's to know files on disk so that it could move them to remote sites for execution or as inputs to applications. Mapped file variables also indicate to swift that, when files are created on remote sites, they need to be staged back. Swift/K provides several mappers which makes it convenient to map files on disk to file variables.

There are two choices here :

1. Have the user define the mappers and data objects
2. Have the data objects be created only by Apps.

In Swift, the user defines file mappings like this :

```
# Mapping a single file
file f <"f.txt">;

# Array of files
file texts[] <filesys_mapper; prefix="foo", suffix=".txt">;
```

The files mapped to an array could be either inputs or outputs to be created. Which is the case is inferred from whether they are on the left-hand side or right-hand side of an assignment. Variables on the left-hand side are inferred to be outputs that have future-like behavior. To avoid conflicting values being assigned to the same variable, Swift variables are all immutable.

For instance, the following would be a major concern *if* variables were not immutable:

```
x = 0;
x = 1;
trace(x);
```

The results that trace would print would be non-deterministic, if x were mutable. In Swift, the above code would raise an error. However this is perfectly legal in python, and the x would take the last value it was assigned.

9.1.3 Remote-Execution

In Swift/K, remote execution is handled by coasters. This is a pilot mechanism that supports dynamic resource provisioning from cluster managers such as PBS, Slurm, Condor and handles data transport from the client to the workers. Swift/T on the other hand is designed to run as an MPI job on a single HPC resource. Swift/T utilized shared-filesystems that almost every HPC resource has.

To be useful, Parsl will need to support remote execution and file transfers. Here we will discuss just the remote-execution aspect.

Here is a set of features that should be implemented or borrowed :

- [Must have] New remote execution system must have the executor interface.
- [Must have] Executors must be memory efficient wrt to holding jobs in memory.
- [Must have] Continue to support both BashApps and PythonApps.
- [?] Capable of using templates to submit jobs to Cluster resource managers.
- [?] Dynamically launch and shutdown workers.

Note: Since the current roadmap to remote execution is through ipython-parallel, we will limit ourselves to Python3.5 to avoid library naming issues.

9.1.4 Availability of Python3.5 on target resources

The availability of Python3.5 on compute resources, especially one's on which the user does not have admin privileges could be a concern. This was raised by Lincoln from the OSG Team. Here's a small table of our initial target systems as of Mar 3rd, 2017 :

Compute Resource	Python3.4	Python3.5	Python3.6
Midway (RCC, UChicago)	X	X	
Open Science Grid	X	X	
BlueWaters	X	X	
AWS/Google Cloud	X	X	X
Beagle	X		

CHAPTER 10

Roadmap

Sufficient capabilities to use Parsl in many common situations already exist. This document indicates where Parsl is going; it contains a list of features that Parsl has or will have. Features that exist today are marked in bold. Help in providing any of the yet-to-be-developed capabilities is welcome.

10.1 Core Functionality

- **Parsl has the ability to execute standard python code and to asynchronously execute tasks, called Apps.**
  - **Any Python function annotated with "@App" is an App.**
  - **Apps can be Python functions or bash scripts that wrap external applications.**
- **Asynchronous tasks return futures, which other tasks can use as inputs.**
  - **This builds an implicit data flow graph.**
- **Asynchronous tasks can execute locally on threads or as separate processes.**
- **Asynchronous tasks can execute on a remote resource.**
  - **libsubmit (to be renamed) provides this functionality.**
  - **A shared filesystem is assumed; data staging (of files) is not yet supported.**
- **The Data Flow Kernel (DFK) schedules Parsl task execution (based on dataflow).**

10.2 Data management

- **File abstraction to support representation of local and remote files.**
- Support for a variety of common data access protocols (e.g., **local**, HTTP, Globus).

- Input/output staging models that support transparent movement of data from source to a location on which it is accessible for compute. This includes staging to/from the client (script execution location), service (pilot job controller location), and worker node.
- Support for creation of a sandbox and execution within this environment for systems without a shared file system, to isolate data and simplify script code. Sandbox execution can optionally be turned off in extreme scale environments.
- Support for data caching at multiple levels and across sites.

10.3 Execution core and parallelism (DFK)

- **Support for application and data futures within scripts**
- **Internal (dynamically created/updated) task/data dependency graph that enables asynchronous execution ordered by data dependencies and throttled by resource limits**
- Well defined state transition model for task lifecycle
- More efficient algorithms for managing dependency resolution.
- Scheduling and allocation algorithms that determine job placement based on job and data requirements (including deadlines) as well as site capabilities
- Logic to manage (provision, resize) execution resource block based on job requirements, and fitting tasks into the resource blocks
- Retry logic to support recovery and fault tolerance

10.4 Resource provisioning and execution

- **Uniform abstraction for execution resources (to support resource provisioning, job submission, allocation management) on cluster, cloud, and supercomputing resources**
- **Support for different execution models on any execution provider (e.g., pilot jobs using Ipython parallel on clusters and extreme-scale execution using Swift/T on supercomputers)**
- Cloud-hosted site configuration repository that stores configurations for resource authentication, data staging, and job submission endpoints
- API/method for {adding entries to, viewing entries} in repository
- Support for remote execution using **SSH** and OAuth-based authentication
- Utilizing multiple sites for a single script's execution
- IPP workers to support multiple threads of execution per node.
- Support for user-defined containers as Parsl apps and orchestration of workflows comprised of containers

10.5 Visualization, debugging, fault tolerance

- **Support for exception handling**
- Interface for accessing real-time state and audit logs

- Visualization library that enables users to introspect graph, task, and data dependencies, as well as observe state of executed/executing tasks
- Integration of visualization into jupyter
- Support for visualizing dead/dying parts of the task graph and retrying with updates to the task.
- Retry model to selectively re-execute only the failed branches of a workflow graph
- Fault tolerance support for individual task execution

10.6 Authentication and authorization

- Seamless authentication using OAuth-based methods within Parsl scripts (e.g., native app grants)
- Support for arbitrary identity providers and pass through to execution resources (including 2FA)
- Support for transparent/scoped access to external services (e.g., Globus transfer)

10.7 Ecosystem

- Support for CWL, ability to execute CWL workflows and use CWL app descriptions
- Creation of library of Parsl apps and workflows
- Provenance capture/export in standard formats
- Automatic metrics capture and reporting to understand Parsl usage

10.8 Documentation / Tutorials:

- Documentation about Parsl and its features
- Documentation about supported sites and how to use them
- Self-guided Jupyter notebook tutorials on Parsl features
- Hands-on tutorial suitable for webinars and meetings

CHAPTER 11

Developer Documentation

11.1 Parsl

Parallel Scripting Library, designed to enable efficient workflow execution.

11.1.1 Importing

To get all the required functionality, we suggest importing the library as follows:

```
>>> import parsl
>>> from parsl import *
```

11.1.2 Logging

Following the general logging philosophy of python libraries, by default Parsl doesn't log anything. However the following helper functions are provided for logging:

1. **set_stream_logger** This sets the logger to the StreamHandler. This is quite useful when working from a Jupyter notebook.
2. **set_file_logger** This sets the logging to a file. This is ideal for reporting issues to the dev team.

parsl.**set_stream_logger**(*name='parsl'*, *level=10*, *format_string=None*)
Add a stream log handler

Args:

- name (string) : Set the logger name.
- level (logging.LEVEL) : Set to logging.DEBUG by default.
- format_string (sting) : Set to None by default.

Returns:

- None

parsl.**set_file_logger**(*filename*, *name='parsl'*, *level=10*, *format_string=None*)
Add a stream log handler

Args:

- filename (string): Name of the file to write logs to
- name (string): Logger name
- level (logging.LEVEL): Set the logging level.
- format_string (string): Set the format string

Returns:

- None

11.2 Apps

Apps are parallelized functions that execute independent of the control flow of the main python interpretor. We have two main types of Apps : PythonApps and BashApps. These are subclassed from AppBase.

11.2.1 AppBase

This is the base class that defines the two external facing functions that an App must define. The __init__ () which is called when the interpretor sees the definition of the decorated function, and the __call__ () which is invoked when a decorated function is called by the user.

class parsl.app.app.**AppBase**(*func*, *executor*, *walltime=60*, *exec_type='bash'*)
This is the base class that defines the two external facing functions that an App must define. The __init__ () which is called when the interpretor sees the definition of the decorated function, and the __call__ () which is invoked when a decorated function is called by the user.

__call__(**args*, ***kwargs*)
The __call__ function must be implemented in the subclasses

__init__(*func*, *executor*, *walltime=60*, *exec_type='bash'*)
Constructor for the APP object.

Args:

- func (function): Takes the function to be made into an App
- executor (executor): Executor for the execution resource

Kwargs:

- walltime (int) : Walltime in seconds for the app execution
- exec_type (string) : App type (bash|python)

Returns:

- APP object.

11.2.2 PythonApp

Concrete subclass of AppBase that implements the Python App functionality.

class parsl.app.app.**PythonApp**(*func*, *executor*, *walltime=60*)
Extends AppBase to cover the Python App

__call__(**args*, ***kwargs*)
This is where the call to a python app is handled

Args:

- Arbitrary

Kwargs:

- Arbitrary

Returns:

If outputs=[...] was a kwarg then: App_fut, [Data_Futures...]

else: App_fut

__init__(*func*, *executor*, *walltime=60*)
Initialize the super. This bit is the same for both bash & python apps.

11.2.3 BashApp

Concrete subclass of AppBase that implements the Bash App functionality.

class parsl.app.app.**BashApp**(*func*, *executor*, *walltime=60*)

__call__(**args*, ***kwargs*)
This is where the call to a Bash app is handled

Args:

- Arbitrary

Kwargs:

- Arbitrary

Returns:

If outputs=[...] was a kwarg then: App_fut, [Data_Futures...]

else: App_fut

11.3 Futures

Futures are returned as proxies to a parallel execution initiated by a call to an App. We have two kinds of futures in Parsl: AppFutures and DataFutures.

11.3.1 AppFutures

class parsl.dataflow.futures.**AppFuture**(*parent*, *tid=None*)
An AppFuture points at a Future returned from an Executor

We are simply wrapping a AppFuture, and adding the specific case where, if the future is resolved i.e file exists, then the DataFuture is assumed to be resolved.

done()
Check if the future is done. If a parent is set, we return the status of the parent. else, there is no parent assigned, meaning the status is False.

Returns:

- True : If the future has successfully resolved.
- False : Pending resolution

parent_callback(*executor_fu*)
Callback from executor future to update the parent.

Args:

- executor_fu (Future): Future returned by the executor along with callback

Returns:

- None

Updates the super() with the result() or exception()

update_parent(*fut*)
Handle the case where the user has called result on the AppFuture before the parent exists. Add a callback to the parent to update the state

11.3.2 DataFutures

class parsl.app.futures.**DataFuture**(*fut*, *file_obj*, *parent=None*, *tid=None*)
A datafuture points at an AppFuture

We are simply wrapping a AppFuture, and adding the specific case where, if the future is resolved i.e file exists, then the DataFuture is assumed to be resolved.

cancel()
Cancel the task that this DataFuture is tracking.

Note: This may not work

filename
Filepath of the File object this datafuture represents

filepath
Filepath of the File object this datafuture represents

parent_callback(*parent_fu*)
Callback from executor future to update the parent.

Args:

- executor_fu (Future): Future returned by the executor along with callback

Returns:

- None

Updates the super() with the result() or exception()

result (*timeout=None*)
A blocking call that returns either the result or raises an exception. Assumptions : A DataFuture always has a parent AppFuture. The AppFuture does callbacks when setup.

Kwargs:

- timeout (int): Timeout in seconds

Returns:

- If App completed successfully returns the filepath.

Raises:

- Exception raised by app if failed.

tid
Returns the task_id of the task that will resolve this DataFuture

11.4 Exceptions

class parsl.app.errors.**ParslError**
Base class for all exceptions

Only to be invoked when only a more specific error is not available.

class parsl.app.errors.**NotFutureError**
Basically a type error. A non future item was passed to a function that expected a future.

class parsl.app.errors.**InvalidAppTypeError**
An invalid app type was requested from the the @App decorator.

class parsl.app.errors.**AppException**
An error raised during execution of an app. What this exception contains depends entirely on context

class parsl.app.errors.**AppBadFormatting** (*reason*, *exitcode*, *retries=None*)
An error raised during formatting of a bash function What this exception contains depends entirely on context Contains: reason(string) exitcode(int) retries(int/None)

class parsl.app.errors.**AppFailure** (*reason*, *exitcode*, *retries=None*)
An error raised during execution of an app. What this exception contains depends entirely on context Contains: reason(string) exitcode(int) retries(int/None)

class parsl.app.errors.**MissingOutputs** (*reason*, *outputs*)
Error raised at the end of app execution due to missing output files

Contains: reason(string) outputs(List of strings/files..)

class parsl.app.errors.**DependencyError** (*dependent_exceptions*, *reason*, *outputs*)
Error raised at the end of app execution due to missing output files

Contains: reason(string) outputs(List of strings/files..)

class parsl.dataflow.error.**DataFlowExceptions**
Base class for all exceptions Only to be invoked when only a more specific error is not available.

class parsl.dataflow.error.**DuplicateTaskError**
Raised by the DataFlowKernel when it finds that a job with the same task-id has been launched before.

class parsl.dataflow.error.**MissingFutError**
Raised when a particular future is not found within the dataflowkernel's datastructures. Deprecated.

11.5 Executors

Executors are abstractions that represent available compute resources to which you could submit arbitrary App tasks. An executor initialized with an Execution Provider can dynamically scale with the resources requirements of the workflow.

We currently have thread pools for local execution, remote workers from ipyparallel for executing on high throughput systems such as campus clusters, and a Swift/T executor for HPC systems.

11.5.1 ParslExecutor

class parsl.executors.base.**ParslExecutor**

Define the strict interface for all Executor classes This is a metaclass that only enforces concrete implementations of functionality by the child classes.

Note: Shutdown is currently missing, as it is not yet supported by some of the executors (threads for eg).

__init__

Initialize self. See help(type(self)) for accurate signature.

scale_in(**args*, ***kwargs*)

Scale in method. We should have the scale in method simply take resource object which will have the scaling methods, scale_in itself should be a corinine, since scaling tasks can be slow.

scale_out(**args*, ***kwargs*)

Scale out method. We should have the scale out method simply take resource object which will have the scaling methods, scale_out itself should be a coroutine, since scaling tasks can be slow.

scaling_enabled

The callers of ParslExecutors need to differentiate between Executors and Executors wrapped in a resource provider

submit(**args*, ***kwargs*)

We haven't yet decided on what the args to this can be, whether it should just be func, args, kwargs or be the partially evaluated fn

11.5.2 ThreadPoolExecutor

class parsl.executors.threads.**ThreadPoolExecutor**(*max_workers=2*, *thread_name_prefix=''*, *execution_provider=None*, *config=None*)

The thread pool executor

__init__(*max_workers=2*, *thread_name_prefix=''*, *execution_provider=None*, *config=None*)

Initialize the thread pool Config options that are really used are :

config.sites.site.execution.options = {"maxThreads" [<int>,] "threadNamePrefix" : <string>}

Kwargs:

- max_workers (int) : Number of threads (Default=2) (keeping name workers/threads for backward compatibility)
- thread_name_prefix (string) : Thread name prefix (Only supported in python v3.6+
- execution_provider (ep object) : This is ignored here

- config (dict): The config dict object for the site:

scale_in(*workers=1*)
Scale in the number of active workers by 1 This method is notImplemented for threads and will raise the error if called.

Raises: NotImplemented exception

scale_out(*workers=1*)
Scales out the number of active workers by 1 This method is notImplemented for threads and will raise the error if called.

Raises: NotImplemented exception

submit(**args*, ***kwargs*)
Submits work to the thread pool This method is simply pass through and behaves like a submit call as described here Python docs:

Returns: Future

11.5.3 IPyParallelExecutor

class parsl.executors.threads.**ThreadPoolExecutor**(*max_workers=2*, *thread_name_prefix=''*, *execution_provider=None*, *config=None*)
The thread pool executor

__init__(*max_workers=2*, *thread_name_prefix=''*, *execution_provider=None*, *config=None*)
Initialize the thread pool Config options that are really used are :

config.sites.site.execution.options = {"maxThreads" [<int>,] "threadNamePrefix" : <string>}

Kwargs:

- max_workers (int) : Number of threads (Default=2) (keeping name workers/threads for backward compatibility)
- thread_name_prefix (string) : Thread name prefix (Only supported in python v3.6+
- execution_provider (ep object) : This is ignored here
- config (dict): The config dict object for the site:

scale_in(*workers=1*)
Scale in the number of active workers by 1 This method is notImplemented for threads and will raise the error if called.

Raises: NotImplemented exception

scale_out(*workers=1*)
Scales out the number of active workers by 1 This method is notImplemented for threads and will raise the error if called.

Raises: NotImplemented exception

submit(**args*, ***kwargs*)
Submits work to the thread pool This method is simply pass through and behaves like a submit call as described here Python docs:

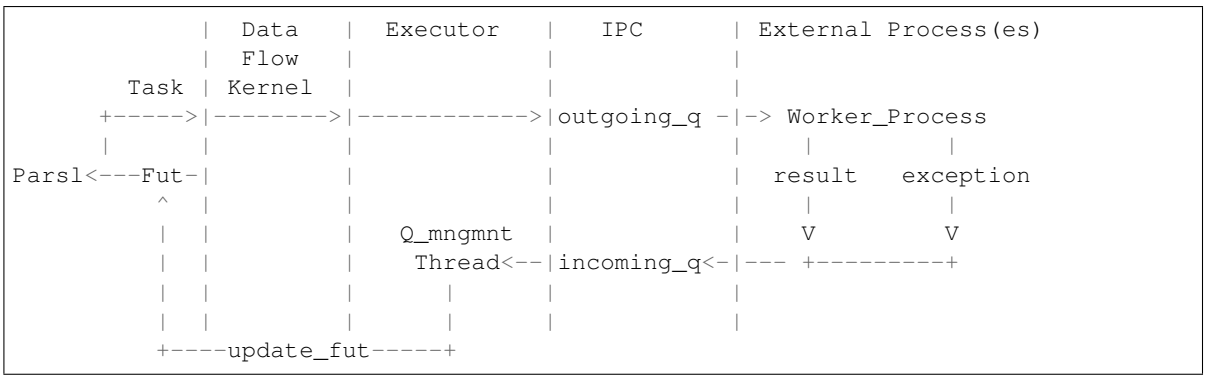
Returns: Future

11.5.4 Swift/Turbine Executor

class parsl.executors.swift_t.**TurbineExecutor**(*swift_attribs=None*)

The Turbine executor. Bypass the Swift/T language and run on top off the Turbine engines in an MPI environment.

Here's a simple diagram



__init__(*swift_attribs=None*)

Initialize the thread pool Trying to implement the emews model.

Kwargs:

- swift_attribs : Takes a dict of swift attribs. Fot future.

_queue_management_worker()

The queue management worker is responsible for listening to the incoming_q for task status messages and updating tasks with results/exceptions/updates

It expects the following messages:

{ "task_id" : <task_id> "result" : serialized result object, if task succeeded ... more tags could be added later

}

{ "task_id" : <task_id> "exception" : serialized exception object, on failure

}

We don't support these yet, but they could be added easily as heartbeat.

{ "task_id" : <task_id> "cpu_stat" : <> "mem_stat" : <> "io_stat" : <> "started" : tstamp

}

The None message is a die request. None

_start_queue_management_thread()

Method to start the management thread as a daemon. Checks if a thread already exists, then starts it. Could be used later as a restart if the management thread dies.

scale_in(*workers=1*)

Scale in the number of active workers by 1 This method is notImplemented for threads and will raise the error if called.

Raises: NotImplemented exception

scale_out(*workers=1*)

Scales out the number of active workers by 1 This method is notImplemented for threads and will raise the error if called. This would be nice to have, and can be done

Raises: NotImplemented exception

shutdown()
Shutdown method, to kill the threads and workers.

submit(*func*, **args*, ***kwargs*)
Submits work to the the outgoing_q, an external process listens on this queue for new work. This method is simply pass through and behaves like a submit call as described here Python docs:

Args:

- func (callable) : Callable function
- *args (list) : List of arbitrary positional arguments.

Kwargs:

- **kwargs (dict) : A dictionary of arbitrary keyword args for func.

Returns: Future

parsl.executors.swift_t.**runner**(*incoming_q*, *outgoing_q*)
This is a function that mocks the Swift-T side. It listens on the the incoming_q for tasks and posts returns on the outgoing_q

Args:

- incoming_q (Queue object) : The queue to listen on
- outgoing_q (Queue object) : Queue to post results on

The messages posted on the incoming_q will be of the form :

{ "task_id" : <uuid.uuid4 string>, "buffer" : serialized buffer containing the fn, args and kwargs

}

If `None` is received, the runner will exit.

Response messages should be of the form:

{ "task_id" : <uuid.uuid4 string>, "result" : serialized buffer containing result "exception" : serialized exception object

}

On exiting the runner will post `None` to the outgoing_q

11.6 Execution Providers

Execution providers are responsible for managing execution resources with a Local Resource Manager (LRM). For instance, campus clusters and supercomputers generally have schedulers such as Slurm, PBS, Condor and. Clouds on the other hand have API interfaces that allow much more fine grain composition of an execution environment. An execution provider abstracts these resources and provides a single uniform interface to them.

11.6.1 ExecutionProvider

11.6.2 Slurm

11.6.3 Amazon Web Services

11.6.4 Azure

CHAPTER 12

Packaging

Currently packaging is managed by Yadu.

Here are the steps:

```
# Create a new git tag :
git tag <MAJOR>.<MINOR>.<BUG_REV>
# Push tag to github :
git push origin <TAG_NAME>

# Depending on permission all of the following might have to be run as root.
sudo su

# Make sure to have twine installed
pip3 install twine

# Create a source distribution
python3 setup.py sdist

# Create a wheel package, which is a prebuilt package
python3 setup.py bdist_wheel

# Upload the package with twine
twine upload dist/*
```

CHAPTER 13

Indices and tables

- genindex
- modindex
- search

Python Module Index

p

Index

Symbols

A

B

C

D

F

I

M

N

P

R

S

T

U