Parsl Documentation

Release 0.5.1

Yadu Nand Babuji

Jun 25, 2018

Contents

Parsl is a Python-based parallel scripting library that supports development and execution of asynchronous and parallel data-oriented workflows (dataflows). These workflows glue together existing executables (called Apps) and Python functions with control logic written in Python. Parsl brings implicit parallel execution to standard Python scripts. Rather than explicitly defining a graph and/or modifying data structures, instead developers simply annotate 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 then executes these components when dependencies are met. Parsl is resource-independent, that is, the same Parsl script can be executed on your laptop through to clusters, clouds, and supercomputers. Parsl also supports different executors including local threads, pilot jobs, and extreme-scale execution with Swift/T.

Parsl can be used to realize a variety of workflows:

- Parallel task-based workflows in which tasks are executed when their dependencies are met.
- Interactive and dynamic workflows in which the workflow is dynamically expanded during execution by users or the workflow itself.
- Procedural workflows in which serial execution of tasks are managed by Parsl.
- Workflows with many short duration tasks where no task-level fault tolerance is required
- Workflows with long running tasks with fault tolerance

Note: By default, Parsl collects anonymous usage statistics for reporting and improvement purposes. To understand what stats are collected and to disable collection please refer to the usage tracking guide

---

CHAPTER 1

Quickstart

To try Parsl now (without installing any code) experiment with our hosted tutorial notebooks

1.1 Installation

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

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

Parsl has been tested on Linux and MacOS.

1.1.1 Installation using Pip

While `pip` and `pip3` can be used to install Parsl we suggest the following approach for reliable installation when many Python environments are avaialble.

1. Install Parsl:

```
$ python3 -m pip install parsl

(to update a previously installed parsl to a newer version, use: python3 -m pip␣
↪install -U 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 Installation using Conda

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

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

2. Install Parsl:

```
$ python3 -m pip install parsl

(to update a previously installed parsl to a newer version, use: python3 -m pip␣
↪install -U parsl)
```

1.2 For Developers

1. Download Parsl:

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

2. Install:

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

3. Use Parsl!

1.3 Requirements

Parsl requires the following :

- Python 3.5+

For testing:

- nose
- coverage

For building documentation:

- nbsphinx
- sphinx
- sphinx_rtd_theme

CHAPTER 2

Parsl tutorial

Parsl is a native Python library that allows you to write functions that execute in parallel and tie them together with dependencies to create workflows. Parsl wraps Python functions as "Apps" using the **@App** decorator. Decorated functions can run in parallel when all their inputs are ready.

For more comprehensive documentation and examples, please refer our documentation.

```
In [ ]: import parsl
        from parsl import *
        # parsl.set_stream_logger() # <-- log everything to stdout
        print(parsl.__version__)
```

2.1 DataFlowKernel

Parsl's `DataFlowKernel` acts as an abstraction layer over any pool of execution resources (e.g., clusters, clouds, threads).

We'll come back to the DataFlowKernel later in this tutorial. For now, we configure this example to use a pool of threads to facilitate local parallel execution.

```
In [ ]: local_config = {
            "sites" : [
                { "site" : "Threads",
                  "auth" : { "channel" : None },
                  "execution" : {
                      "executor" : "threads",
                      "provider" : None,
                      "maxThreads" : 4
                  }
                }],
            "globals" : {"lazyErrors" : True}
        }

        dfk = DataFlowKernel(config=local_config)
```

2.1.1 Apps

In Parsl an `app` is a piece of code that can be asynchronously executed on an execution resource (e.g., cloud, cluster, or local PC). Parsl provides support for pure Python apps and also command-line apps executed via Bash.

2.2 Python Apps

As a first example let's define a simple Python function that returns the string 'Hello World!'. This function is made into a Parsl App using the **@App** decorator. The decorator specifies the type of App ('python'|'bash') and the `DataFlowKernel` object as arguments.

```
In [ ]: @App('python', dfk)
        def hello ():
            return 'Hello World!'

        print(hello().result())
```

As can be seen above, Apps wrap standard Python function calls. As such, they can be passed arbitrary arguments and return standard Python objects.

```
In [ ]: @App('python', dfk)
        def multiply (a, b):
            return a * b

        print(multiply(5,9).result())
```

2.3 Bash Apps

Parsl's Bash app allows you to wrap execution of external applications from the command-line as you would in a Bash shell. It can also be used to execute Bash scripts directly. To define a Bash app the wrapped Python function must return the command-line string to be executed.

Parsl is able to capture stdout/stderr for debugging or as a first class data object in a workflow.

```
In [ ]: @App('bash', dfk)
        def echo_hello(stdout='echo-hello.stdout', stderr='echo-hello.stderr'):
            return 'echo "Hello World!"'

        echo_hello().result()

        with open('echo-hello.stdout', 'r') as f:
             print(f.read())
```

Often, Parsl Apps exchange data in the form of files. In order to orchestrate a dataflow it is important that Parsl is able to track the data that is passed into and out of an App. For this purpose Parsl Apps can define input and output files as follows.

We first create three test files named hello1.txt, hello2.txt, and hello3.txt containing the text "hello 1", "hello 2", and "hello 3".

```
In [ ]: !echo "hello 1" > /tmp/hello1.txt
        !echo "hello 2" > /tmp/hello2.txt
        !echo "hello 3" > /tmp/hello3.txt
```

We then write an App that will concentate these files using `cat`. We pass in the list of hello files and concatenate the text into an output file named all_hellos.txt.

```
In [ ]: @App('bash', dfk)
        def cat(inputs=[], outputs=[]):
            return 'cat %s > %s' %(inputs[0], outputs[0])

        concat = cat(inputs=['/tmp/hello*.txt'], outputs=['all_hellos.txt'])

        # Get the filepath for the output file and open it
        with open(concat.outputs[0].result(), 'r') as f:
             print(f.read())
```

2.3.1 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 it may not be desirable to wait for completion, instead it is often preferable that functions are asynchronous. Parsl provides such asynchronous behavior by returning a future in lieu of results. A future is essentially an object that allows us to track the status of an asynchronous task so that it may, in the future, be interrogated to find the status, results, exceptions, etc.

Parsl provides two types of futures: AppFutures and DataFutures. While related, these two types of futures enable subtly different workflow patterns, as we will see.

2.4 AppFutures

AppFutures are the basic building block upon which Parsl scripts are built. Every invocation of a Parsl app returns an AppFuture which may be used to manage execution and control the workflow.

Here we show how AppFutures are used to wait for the result of a Python App.

```
In [ ]: # App that sleeps and then returns hello world
        @App('python', dfk)
        def hello ():
            import time
            time.sleep(5)
            return 'Hello World!'

        app_future = hello()

        # Check if the app_future is resolved
        print ('Done: %s' % app_future.done())

        # Print the result of the app_future. Note: this
        # call will block and wait for the future to resolve
        print ('Result: %s' % app_future.result())
        print ('Done: %s' % app_future.done())
```

2.5 DataFutures

While AppFutures represent the execution of an asynchronous app, the DataFuture represents the files it produces. Parsl's dataflow model, in which data flows from one app to another via files, requires such a construct to enable apps to validate creation of required files and to subsequently resolve dependencies when input files are created. When invoking an app, Parsl requires that a list of output files be specified (using the outputs keyword argument). A DataFuture for each file is returned by the app when it is executed. Throughout execution of the app Parsl will monitor these files to 1) ensure they are created, and 2) pass them to any dependent apps.

```
In [ ]: # App that echos the input message to the first file specified in the
        # outputs list
        @App('bash', dfk)
        def slowecho(message, outputs=[]):
            return 'sleep 5; echo %s &> {outputs[0]}' % (message)

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

        # The AppFuture's outputs attribute is a list of DataFutures
        print(hello.outputs)

        # Also check the AppFuture
        print ('Done: %s' % hello.done())

        # Print the contents of the output DataFuture when complete
        with open(hello.outputs[0].result(), 'r') as f:
             print(f.read())

        # Now that this is complete, check the DataFutures again, and the Appfuture
        print(hello.outputs)
        print ('Done: %s' % hello.done())
```

2.5.1 Data Management

Parsl is designed to enable implementation of dataflow patterns. These patterns enable workflows to be defined in which the data passed between apps manages the flow of execution. Dataflow programming models are popular as they can cleanly express, via implicit parallelism, the concurrency needed by many applications in a simple and intuitive way.

2.6 Files

Parsl's file abstraction abstracts local access to a file. It therefore requires only the file path to be defined. Irrespective of where the script, or its apps are executed, Parsl uses this abstraction to access that file. When referencing a Parsl file in an app, Parsl maps the object to the appropriate access path.

```
In [ ]: from parsl.data_provider.files import File

        # App that copies the contents of 1 or more files to another file
        @App('bash', dfk)
        def copy(inputs=[], outputs=[]):
             return 'cat %s &> %s' % (inputs[0], outputs[0])

        # cCeate a test file
        open('cat-in.txt', 'w').write('Hello World!\n')

        # Create Parsl file objects
        parsl_infile = File("cat-in.txt")
        parsl_outfile = File("cat-out.txt")

        # Call the copy app with the Parsl file
        copy_future = copy(inputs=[parsl_infile], outputs=[parsl_outfile])

        # Read what was redirected to the output file
```

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

2.6.1 Composing a workflow

Now that we understand all the building blocks, we can create workflows with Parsl. Unlike other workflow systems, Parsl creates implicit workflows based on the passing of control or data between Apps. The flexibility of this model allows for the creation of a wide range of workflows from sequential through to complex nested, parallel workflows. As we will see below, a range of workflows can be created by passing AppFutures and DataFutures between Apps.

2.7 Sequential workflow

Simple sequential or procedural workflows can be created by passing an AppFuture from one task to another. The following example shows one such workflow, which first generates a random number and then writes it to a file.

```
In [ ]: # App that generates a random number
        @App('python', dfk)
        def generate(limit):
              from random import randint
              return randint(1,limit)

        # App that writes a message to a file
        @App('bash', dfk)
        def save(message, outputs=[]):
              return 'echo %s &> {outputs[0]}' % (message)

        # Generate the random number
        message = generate(10)
        print('Random number: %s' % message.result())

        # Save the random number to a file
        saved = save(message, outputs=['output.txt'])

        # Print the output file
        with open(saved.outputs[0].result(), 'r') as f:
              print('File contents: %s' % f.read())
```

2.8 Parallel workflow

The most common way that Parsl Apps are executed in parallel is via looping. The following example shows how a simple loop can be used to create many random numbers in parallel.

```
In [ ]: # App that generates a random number
        @App('python', dfk)
        def generate(limit):
            from random import randint
            return randint(1,limit)

        # Generate 5 random numers
        rand_nums = []
        for i in range(5):
            rand_nums.append(generate(10))

        # Wait for all apps to finish and collect the results
```

```
outputs = [i.result() for i in rand_nums]

# Print results
print(outputs)
```

2.9 Parallel dataflow

Parallel dataflows can be developed by passing data between Apps. In this example we create a set of files, each with a random number, we then concatenate these files into a single file and compute the sum of all numbers in that file. In the first two Apps files are exchanged. The final App returns the sum as a Python integer.

```
In [ ]: # App that generates a random number
        @App('bash', dfk)
        def generate(outputs=[]):
            return "echo $(( RANDOM )) &> {outputs[0]}"

        # App that concatenates input files into a single output file
        @App('bash', dfk)
        def concat(inputs=[], outputs=[], stdout="stdout.txt", stderr='stderr.txt'):
            return "cat {0} > {1}".format(" ".join(inputs), outputs[0])

        # App that calculates the sum of values in a list of input files
        @App('python', dfk)
        def total(inputs=[]):
            total = 0
            with open(inputs[0], 'r') as f:
                for l in f:
                    total += int(l)
            return total

        # Create 5 files with random numbers
        output_files = []
        for i in range (5):
             output_files.append(generate(outputs=['random-%s.txt' % i]))

        # Concatenate the files into a single file
        cc = concat(inputs=[i.outputs[0] for i in output_files], outputs=["all.txt"])

        # Calculate the sum of the random numbers
        total = total(inputs=[cc.outputs[0]])
        print (total.result())
```

2.9.1 Examples

2.10 Monte Carlo workflow

Many scientific applications use the monte-carlo method to compute results.

If a circle with radius is inscribed inside a square with side length then the area of the circle is and the area of the square is . Thus, if uniformly distributed random points are dropped within the suqare then approximately will be inside the circle.

Each call to the function `pi()` is executed independently and in parallel. The `avg_three()` app is used to compute the average of the futures that were returned from the `pi()` calls.

The dependency chain looks like this:

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

```
In [ ]: # App that estimates pi by placing points in a box
        @App('python', dfk)
        def pi(total):
            import random

            # Set the size of the box (edge length) in which we drop random points
            edge_length = 10000
            center = edge_length / 2
            c2  = center ** 2
            count = 0

            for i in range(total):
                # Drop a random point in the box.
                x,y = random.randint(1, edge_length),random.randint(1, edge_length)
                # Count points within the circle
                if (x-center)**2 + (y-center)**2 < c2:
                    count += 1

            return (count*4/total)

        # App that computes the average of the values
        @App('python', dfk)
        def avg_points(a, b, c):
            return (a + b + c)/3

        # Estimate three values for pi
        a, b, c = pi(10**6), pi(10**6), pi(10**6)

        # Compute the average of the three estimates
        avg_pi  = avg_points(a, b, c)

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

2.10.1 Execution and configuration

Parsl is designed to support arbitrary execution providers (e.g., PCs, clusters, supercomputers) and execution models (e.g., threads, pilot jobs, etc.). That is, Parsl scripts are independent of execution provider or executor. Instead, the configuration used to run the script tells Parsl how to execute apps on the desired environment. Parsl provides a high level abstraction, called a Block, for describing the resource configuration for a particular app or script.

Information about the different execution providers and executors supported is included in the Parsl documentation.

2.11 Local execution with threads

As we saw above, we can configure Parsl to execute apps on a local thread pool. This is a good way to parallelize execution on a local PC. The configuration object defines the sites that will be used for execution, optinally the authentication method to be used (e.g., if using SSH), and the execution model to use. In the case of threads we define the maximum number of threads to be used. A number of global configuration options may also be specified.

```
In [ ]: threads_config = {
            "sites" : [
                { "site" : "Local_Threads",
                  "auth" : { "channel" : None },
                  "execution" : {
                      "executor" : "threads",
                      "provider" : None,
                      "maxThreads" : 4
                  }
                }],
            "globals" : {"lazyErrors" : True}
        }
```

2.12 Local execution with pilot jobs

We can also define a configuration that uses IPythonParallel as the executor. In this mode, pilot jobs are used to manage the submission. Parsl creates an IPythonParallel controller to manage execution and deploys one or more IPythonParallel engines (workers) to execute workload. The following config will instantiate this infrastructure locally, it can be trivially extended to include a remote provider (e.g., Cori, Theta, etc.) for execution.

```
In [ ]: ipp_config = {
            "sites" : [{
                "site" : "Local_IPP",
                "auth" : {
                    "channel" : "local"
                },
                "execution" : {
                    "executor" : "ipp",
                    "provider" : "local",
                    "block" : {
                        "nodes" : 1,
                        "taskBlocks" : 1,
                        "walltime" : "00:05:00",
                        "initBlocks" : 1,
                        "minBlocks" : 1,
                        "maxBlocks" : 1,
                        "options" : {
                            "partition" : "debug"
                        }
                    }
                }
            }],
            "globals" : { "lazyErrors" : True },
        }
```

2.13 Running a workflow using a configuration

We can now run the same workflow using either of the two configurations defined above. Change which config is used to instantiate the DFK to see the same workflow executed with different models.

```
In [ ]: import parsl
        from parsl import *
        # parsl.set_stream_logger() # <-- log everything to stdout
        print(parsl.__version__)

        #dfk = DataFlowKernel(config=threads_config)
        dfk = DataFlowKernel(config=ipp_config)

        @App('bash', dfk)
        def generate(outputs=[]):
            return "echo $(( RANDOM )) &> {outputs[0]}"

        @App('bash', dfk)
        def concat(inputs=[], outputs=[], stdout="stdout.txt", stderr='stderr.txt'):
            return "cat {0} > {1}".format(" ".join(inputs), outputs[0])

        @App('python', dfk)
        def total(inputs=[]):
            total = 0
            with open(inputs[0], 'r') as f:
                for l in f:
                    total += int(l)
            return total

        # Create 5 files with random numbers
        output_files = []
        for i in range (5):
             output_files.append(generate(outputs=['random-%s.txt' % i]))

        # Concatenate the files into a single file
        cc = concat(inputs=[i.outputs[0].filepath for i in output_files],
                    outputs=["combined.txt"])

        # Calculate the sum of the random numbers
        total = total(inputs=[cc.outputs[0]])

        print (total.result())
        dfk.cleanup()
```

CHAPTER 3

User guide

3.1 Overview

Parsl is designed to enable the straightforward orchestration of asynchronous tasks into dataflow-based workflows in Python. Parsl manages the parallel execution of these tasks across computation resources when dependencies (e.g., input data dependencies) are met.

Developing a workflow is a two-step process:

1. Annotate functions that can be executed in parallel as Parsl `Apps`.
2. Specify dependencies between functions using standard Python code.

In Parsl, the execution of an `App` yields futures. These futures can be passed to other `Apps` as inputs, establishing a dependency. These dependencies are assembled implicitly into directed acyclic graphs, although these are never explicitly expressed. It is important to note that this graph is dynamically built and then update while the Parsl script executes. That is, the graph is not computed in advanced and is only complete when the script finishes executing. `Apps` that have all their dependencies met are slated for execution (in parallel). This allows Parsl to exploit all parallelism to the fullest extent and at the granularity expressed by the user.

At the heart of Parsl is the DataFlow Kernel (DFK). The DFK is responsible for managing the dynamic graph and determining when tasks can be executed.

A MapReduce job can be simply defined as follows:

```
# Map function that returns double 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
```

(continues on next page)

(continued from previous page)

```
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)

# Reduce phase: apply an *app* function to the set of results
total = app_sum(inputs=mapped_results)

print(total.result())
```

3.2 Apps

In Parsl an "app" is a piece of code that can be asynchronously executed on an execution resource. An execution resource in this context is any target system such as a laptop, cluster, cloud, or even supercomputer. Execution on these resources can be performed by a pool of threads, processes, or remote workers.

Parsl apps are defined by annotating Python functions with the `@App` decorator. Currently two different types of apps can be defined: Python and Bash. Python apps encapsulate pure Python code, while Bash apps wrap calls to external applications and scripts.

3.2.1 Python Apps

The following code snippet shows a simple Python function used to double the input value (`double(Int)`). This function is defined as a Parsl app using the `@App` decorator. The first argument to `@App` specifies the App type as "python". The second argument `dfk` is the Dataflow Kernel which must be configured with appropriate execution resources (e.g., local thread execution).

Python apps are *pure* Python functions. As these functions are executed asynchronously, and potentially remotely, it is important to note that they must explicitly import any required modules and act only on defined input arguments (i.e., it cannot include variables used elsewhere in the script).

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

double(x)
```

Python apps may also act upon files. In order to make these files known to Parsl's Dataflow Kernel you must define the input and output files to be managed. The following code illustrates a simple example that will copy the contents of one file to another.

```
@App('python', dfk)
def echo(inputs=[], outputs=[]):
      with open(inputs[0], 'r') as in_file, open(outputs[0], 'w') as out_file:
          out_file.write(in_file.readline())

echo(inputs=[in.txt], outputs=[out.txt])
```

Limitations

There are limitations on what Python functions can be converted to apps:

1. Functions should act only on defined input arguments.
2. Functions must explicitly import any required modules.
3. Functions should not use script-level or global variables.
4. Parsl uses cloudpickle and pickle to serialize Python constructs, such as inputs and outputs to functions. Therefore, Python apps can only use inputs and outputs that can be serialized by cloudpickle or pickle.

Special Keywords Arguments

Any Parsl app (decorated with the `@App` decorator) can use the following special reserved keyword arguments.

1. inputs : (list) This keyword argument allows you to pass a list of input *Futures*, and thus wait on the results of these futures to be resolved before execution.

2. outputs : (list) This keyword argument allows you to explicitly list the output *Futures* that will be produced by this app. Parsl will track these files and ensure they are correctly created. They can then be passed to other apps as input arguments.

Returns

A python app returns an AppFuture that is a proxy for the results that will be returned by the app once it is executed. This futures itself holds the python object(s) returned by the app. In case of a failure in the app, the future holds the exception raised by the app.

3.2.2 Bash Apps

Parsl's Bash app allows you to wrap execution of external applications from the command-line as you would in a Bash shell. It can also be used to execute Bash scripts directly. To define a Bash app the wrapped Python function must return the command-line string to be executed.

The following code snippet shows a simple Bash script written as a string in Python and wrapped as an app. Any command-line invocation represented by an arbitrarily long string, can be returned by a function decorated within an `@App` of type *bash* to be executed. Unlike the Python app, Bash apps communicate by passing files. The decorated *bash* function provides the same special keyword arguments to manage input and output files. In addition, it also includes keyword arguments for capturing the STDOUT and STDERR streams and recording them in files that are managed by Parsl.

```
@App('bash', dfk)
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()
```

Limitations

The following limitations apply to Bash apps:

1. Environment variables are not yet supported.

Special Keywords

1. inputs: (list) A list of input *Futures* on which to wait before execution.
2. outputs: (list) A list of output *Futures* that will be created by the app.
3. stdout: (string) The path to a file to which STDOUT should be redirected.
4. stderr: (string) The path to a file to which STDERR should be redirected.

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', dfk)
def echo(arg1, inputs=[], stderr='std.err', stdout='std.out'):
    return 'echo %s %s %s' % (arg1, inputs[0], inputs[1])

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

Returns

A bash app returns an AppFuture just like a python app however the values returned by the future are quite different. In Unix fashion, the result made available upon completion is the **return/exit code** of the bash script. This future may also hold various exceptions that capture errors during execution such as incorrect privileges, missing output files etc.

3.3 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 it may not be desirable to wait for completion, instead it is often preferable that functions are asynchronous. Parsl provides such asynchronous behavior by returning a future in lieu of results. A future is essentially an object that allows us to track the status of an asynchronous task so that it may, in the future, be interrogated to find the status, results, exceptions, etc. A future is a proxy for a result that is not yet available.

Parsl provides two types of futures: AppFutures and DataFutures. While related, these two types of futures enable subtly different workflow patterns, as we will see.

3.3.1 AppFutures

AppFutures are the basic building block upon which Parsl scripts are built. Every invocation of a Parsl app returns an AppFuture which may be used to manage execution and control the workflow. AppFutures are inherited from the python's concurrent library. There are several key functionalities provided by AppFutures:

1. An AppFuture provides a way to check the current status of a Parsl app, without waiting for it to complete.

```
@App('python', dfk)
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. An AppFuture provides a way to block and wait for the result of the app:

```
@App('python', dfk)
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() function will block until the app has completed
print(doubled_x.result())
```

3. An AppFuture provides a safe way to handle exceptions and errors while executing complex workflows.

```
@App('python', dfk)
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 ze:
     print('Oops! You tried to divide by 0 ')
except Exception as e:
     print('Oops! Something really bad happened')
```

In addition to being able to capture exceptions raised in a specific app executions, Parsl also raises `DependencyErrors` when apps are unable to execute due to failures in prior dependent apps. That is, an app that is dependent on the successful completion of another app will fail with a dependency error if any of the apps on which it depends fails.

3.3.2 DataFutures

While AppFutures represent the execution of an asynchronous app, the DataFuture represents the files it produces. Parsl's dataflow model, in which data flows from one app to another via files, requires such a construct to enable apps to validate creation of required files and to subsequently resolve dependencies when input files are created. When invoking an app, Parsl requires that a list of output files be specified (using the output keyword argument). A DataFuture for each file is returned by the app when it is executed. Throughout execution of the app Parsl will monitor these files to 1) ensure they are created, and 2) pass them to any dependent apps. DataFutures are accessible through the `outputs` attribute of the AppFuture. DataFuture are inherited from Python's concurrent library.

The following example shows how DataFutures are used:

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

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

# The AppFuture's outputs attribute is a list of DataFutures
print(hello.outputs)

# Print the contents of the output DataFuture when complete
with open(hello.outputs[0].result(), 'r') as f:
     print(f.read())
```

3.4 Composing a workflow

Workflows in Parsl are created implicitly based on the passing of control or data between `Apps`. The flexibility of this model allows for the creation of a wide range of workflows from sequential through to complex nested, parallel workflows. As we will see below a range of workflows can be created by passing AppFutures and DataFutures between `Apps`.

Parsl is also designed to address the requirements of a range of workflows from those that run a large number of very small tasks through to those that run few long running tasks. In each case, Parsl can be configured to optimize deployment towards performance or fault tolerance.

Below we illustrate a range of workflow patterns, however it is important to note that this set of examples is by no means comprehensive.

3.4.1 Procedural workflows

Simple sequential or procedural workflows can be created by passing an AppFuture from one task to another. The following example shows one such workflow which first generates a random number and then writes it to a file. Note: in this case we combine a Pyhton and a Bash `App` seamlessly.

```
# Generate a random number
@App('python', dfk)
def generate(limit):
      from random import randint
      """Generate a random integer and return it"""
      return randint(1,limit)

# write a message to a file
@App('bash', dfk)
def save(message, outputs=[]):
      return 'echo %s &> {outputs[0]}' % (message)

message = generate(10)

saved = save(message, outputs=['output.txt'])
```

(continues on next page)

(continued from previous page)

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

3.4.2 Parallel workflows

Parallel execution occurs automatically in Parsl, respecting dependencies among `App` executions. The following example shows how a single `App` can be used with and without dependencies to demonstrate parallel execution.

```
@App('python', dfk)
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())
```

3.4.3 Parallel workflows with loops

The most common way that Parsl `Apps` are executed in parallel is via looping. The following example shows how a simple loop can be used to create many random numbers in parallel.

```
@App('python', dfk)
def generate(limit):
    from random import randint
    """Generate a random integer and return it"""
    return randint(1,limit)

rand_nums = []
for i in range(1,5):
    rand_nums.append(generate(i))

# wait for all apps to finish and collect the results
outputs = [i.result() for i in rand_nums]
```

3.4.4 Parallel dataflows

Parallel dataflows can be developed by passing data between `Apps`. In this example we create a set of files, each with a random number, we then concatenate these files into a single file and compute the sum of all numbers in that file. In the first two `Apps` files are exchanged. The final `App` returns the sum as a Python integer.

```
@App('bash', dfk)
def generate(outputs=[]):
    return 'echo $(( RANDOM % (10 - 5 + 1 ) + 5 )) &> {outputs[0]}'

@App('bash', dfk)
def concat(inputs=[], outputs=[], stdout='stdout.txt', stderr='stderr.txt'):
    return 'cat {0} >> {1}'.format(' '.join(inputs), outputs[0])

@App('python', dfk)
def total(inputs=[]):
    total = 0
    with open(inputs[0], 'r') as f:
        for l in f:
            total += int(l)
    return total

# create 5 files with random numbers
output_files = []
for i in range (5):
     output_files.append(generate(outputs=['random-%s.txt' % i]))

# concatenate the files into a single file
cc = concat(inputs=[i.outputs[0] for i in output_files], outputs=['all.txt'])

# calculate the average of the random numbers
totals = total(inputs=[cc.outputs[0]])

print (totals.result())
```

3.5 Data Management

Parsl is designed to enable implementation of dataflow patterns in which data passed between apps manages the flow of execution. Dataflow programming models are popular as they can cleanly express, via implicit parallelism, the concurrency needed by many applications in a simple and intuitive way.

Parsl aims to abstract not only parallel execution but also execution location, which in turn requires data location abstraction. This is crucial as it allows scripts to execute in different locations without regard for data location. Parsl implements a simple file abstraction that can be used to reference data irrespective of its location. At present this model supports local files as well as files accessible via Globus. In the near future it will be extended to address remotely accessible files using FTP and HTTP.

3.5.1 Files

The *File* class abstracts the file access layer. Irrespective of where the script or its apps are executed, Parsl uses this abstraction to access that file. When referencing a Parsl file in an app, Parsl maps the object to the appropriate access path according to the selected access *scheme*. Local and Globus schemes are supported, and are described in more detail below.

Local

The *file* scheme is used to reference local files. A file using the local file scheme must specify the absolute file path, for example:

```
File('file://path/filename.txt')
```

The file may then be passed as input or output to an app. Here is an example Parsl script which runs *cat* on a local file:

```
@App('bash', dfk)
def cat(inputs=[], stdout='stdout.txt'):
     return 'cat %s' % (inputs[0])

# create a test file
open('test.txt', 'w').write('Hello\n')

# create the Parsl file
parsl_file = File('file://test.txt')

# call the cat app with the Parsl file
cat(inputs=[parsl_file])
```

Globus

Caution: This feature is available from Parsl `v0.5.0` in an `experimental` state. We request feedback and feature enhancement requests via github.

The *globus* scheme is used to reference files that can be accessed using Globus (a guide to using Globus is available here). A file using the Globus scheme must specify the UUID of the Globus endpoint and a path to the file on the endpoint, for example:

```
File('globus://037f054a-15cf-11e8-b611-0ac6873fc732/unsorted.txt')
```

Note: the Globus endpoint UUID can be found in the Globus Manage Endpoints page.

Like the local file scheme, Globus files may be passed as input or output to a Parsl app. However, in the Globus case, the file object is only an abstract representation of the file on the remote side and thus the file must be staged to or from the execution site. For example, to stage in (transfer a file to the site where the Parsl app will be executed), the `stage_in()` and `result()` functions must be executed explicitly, for example:

```
unsorted_file = File('globus://037f054a-15cf-11e8-b611-0ac6873fc732/unsorted.txt')

dfu = unsorted_file.stage_in()
dfu.result()
```

To stage a file out (transfer a file from the site where the Parsl app is executed), the `stage_out()` and `result()` functions must be executed explicitly, for example:

```
f = sort_strings(inputs=[unsorted_file], outputs=[sorted_file])
f.result()

dfs = sorted_file.stage_out()
dfs.result()
```

Parsl scripts may combine staging of files in and out of apps. For example, the following script stages a file from a remote Globus endpoint, it then sorts the strings in that file, and stages the sorted output file to another remote endpoint.

```
@App('python', dfk)
def sort_strings(inputs=[], outputs=[]):
    with open(inputs[0], 'r') as u:
        strs = u.readlines()
        strs.sort()
        with open(outputs[0].filepath, 'w') as s:
            for e in strs:
                s.write(e)


unsorted_file = File('globus://037f054a-15cf-11e8-b611-0ac6873fc732/unsorted.txt')
sorted_file = File ('globus://ddb59aef-6d04-11e5-ba46-22000b92c6ec/sorted.txt')

dfu = unsorted_file.stage_in()
dfu.result()

f = sort_strings(inputs=[unsorted_file], outputs=[sorted_file])
f.result()

dfs = sorted_file.stage_out()
dfs.result()
```

Configuration

To inform Parsl where the file is to be transferred to or from (i.e., where the Parsl app is executed), the configuration must specify the *endpoint_name* (the UUID of the Globus endpoint that is associated with the system where the parsl app is executed).

In order to manage where data is staged users may configure the default "working_dir" on a site. This is specified in the data configuration object as follows:

```
config = {
    "sites": [
        {
            "data": {
                 "working_dir" : "/home/user/parsl_script"
             }
        }
    ]
}
```

In some cases, for example when using a Globus shared endpoint or when a Globus DTN is mounted on a supercomputer, the path seen by Globus is not the same as the local path seen by Parsl. In this case the configuration may optionally specify a mapping between the *endpoint_path* (the common root path seen in Globus), and the *local_path* (the common root path on the local file system). In most cases *endpoint_path* and *local_path* are the same.

```
config = {
    "sites": [
        {
        ...
            "data": {
                "globus": {
                    "endpoint_name": "7d2dc622-2edb-11e8-b8be-0ac6873fc732",
                    "endpoint_path": "/",
                    "local_path" : "/home/user"
                 }
```

(continues on next page)

(continued from previous page)

```
                }
            }
        ]
}
```

3.6 Execution

Parsl is designed to support arbitrary execution providers (e.g., PCs, clusters, supercomputers) as well as arbitrary execution models (e.g., threads, pilot jobs, etc.). That is, Parsl scripts are independent of execution provider or executor. Instead, the configuration used to run the script tells Parsl how to execute apps on the desired environment. Parsl provides a high level abstraction, called a *Block*, for providing a uniform description of a resource configuration for a particular app or script.

3.6.1 Execution Providers

Execution providers are responsible for managing execution resources. In the simplest case the local computer is used and parallel tasks are forked to individual threads. For larger resources a Local Resource Manager (LRM) is usually used to manage access to resources. For instance, campus clusters and supercomputers generally use LRMs (schedulers) such as Slurm, Torque/PBS, Condor and Cobalt. Clouds, on the other hand, provide APIs that allow more fine-grained composition of an execution environment. Parsl's execution provider abstracts these different resource types and provides a single uniform interface.

Parsl's execution interface is called `libsubmit` (https://github.com/Parsl/libsubmit) – a Python library that provides a common interface to execution providers. Libsubmit defines a simple interface which includes operations such as submission, status, and job management. It currently supports a variety of providers including Amazon Web Services, Azure, and Jetstream clouds as well as Cobalt, Slurm, Torque, GridEngine, and HTCondor LRMs. New execution providers can be easily added by implementing libsubmit's execution provider interface.

3.6.2 Executors

Depending on the execution provider there are a number of ways to then submit workload to that resource. For example, for local execution threads or pilot jobs may be used, for supercomputing resources pilot jobs, various launchers, or even a distributed execution model such as that provided by Swift/T may be used. Parsl supports these models via an *executor* model. Executors represent a particular method via which tasks can be executed. As described below, an executor initialized with an execution provider can dynamically scale with the resources requirements of the workflow.

Parsl currently supports the following executors:

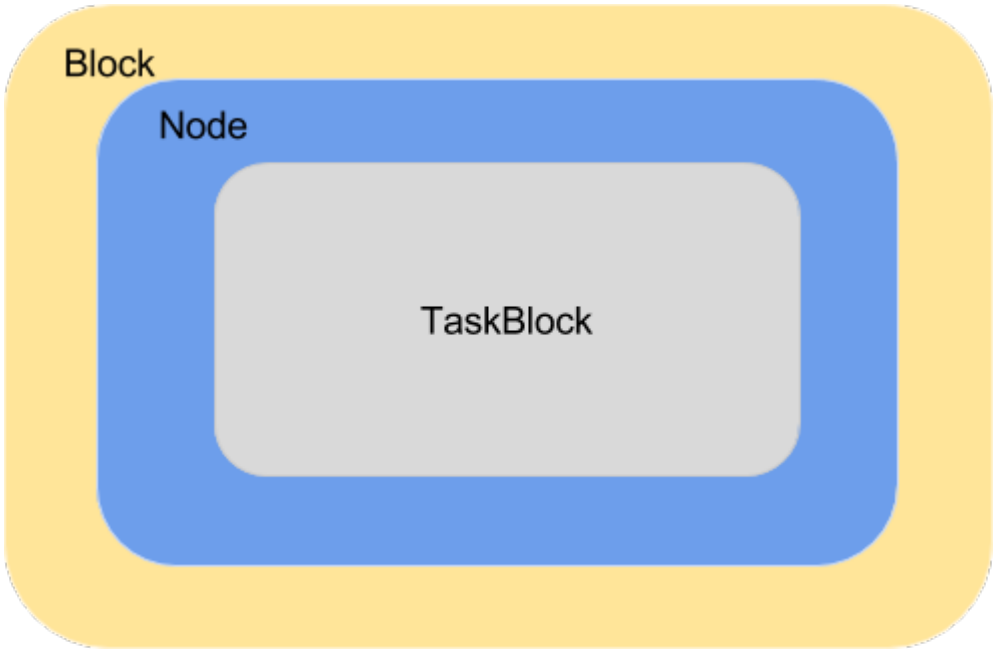
1. **ThreadPoolExecutor**: This executor supports multi-thread execution on local resources.
2. **IPyParallelExecutor**: This executor supports both local and remote execution using a pilot job model. The IPythonParallel controller is deployed locally and IPythonParallel engines are deployed to execution nodes. IPythonParallel then manages the execution of tasks on connected engines.
3. **Swift/TurbineExecutor**: This executor uses the extreme-scale Turbine model to enable distributed task execution across an MPI environment. This executor is typically used on supercomputers.

These executors cover a broad range of execution requirements. As with other Parsl components there is a standard interface (ParslExecutor) that can be implemented to add support for other executors.

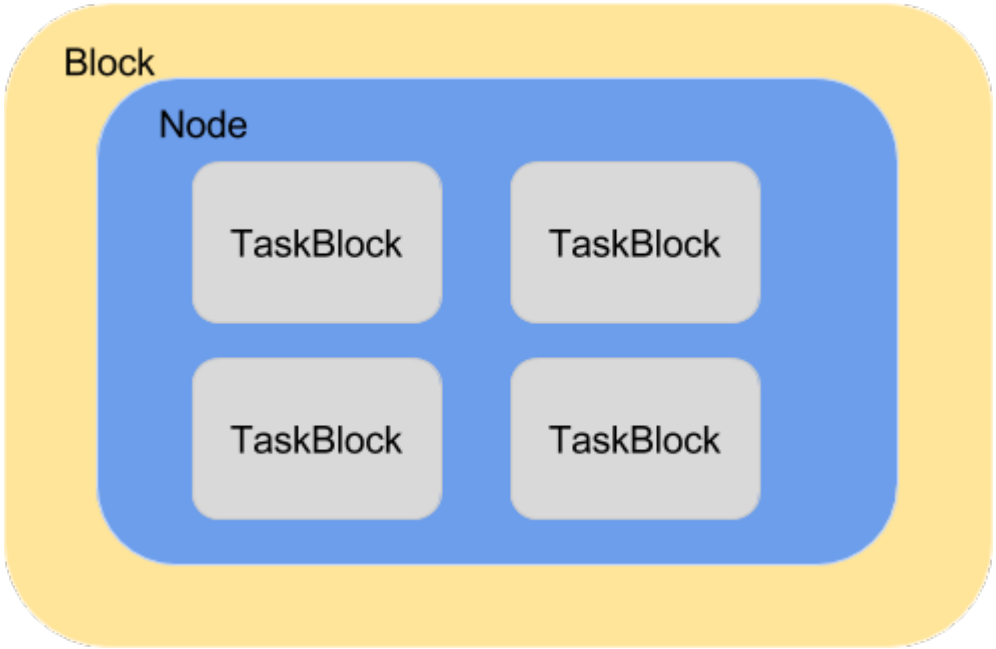
3.6.3 Blocks

Providing a uniform representation of heterogeneous resources is one of the most difficult challenges for parallel execution. Parsl provides an abstraction based on resource units called *blocks*. A block is a single unit of resources that is obtained from an execution provider. Within a block are a number of nodes. Parsl can then create *TaskBlocks* within and across (e.g., for MPI jobs) nodes. A TaskBlock is a virtual suballocation in which individual tasks can be launched. Three different examples of block configurations are shown below.

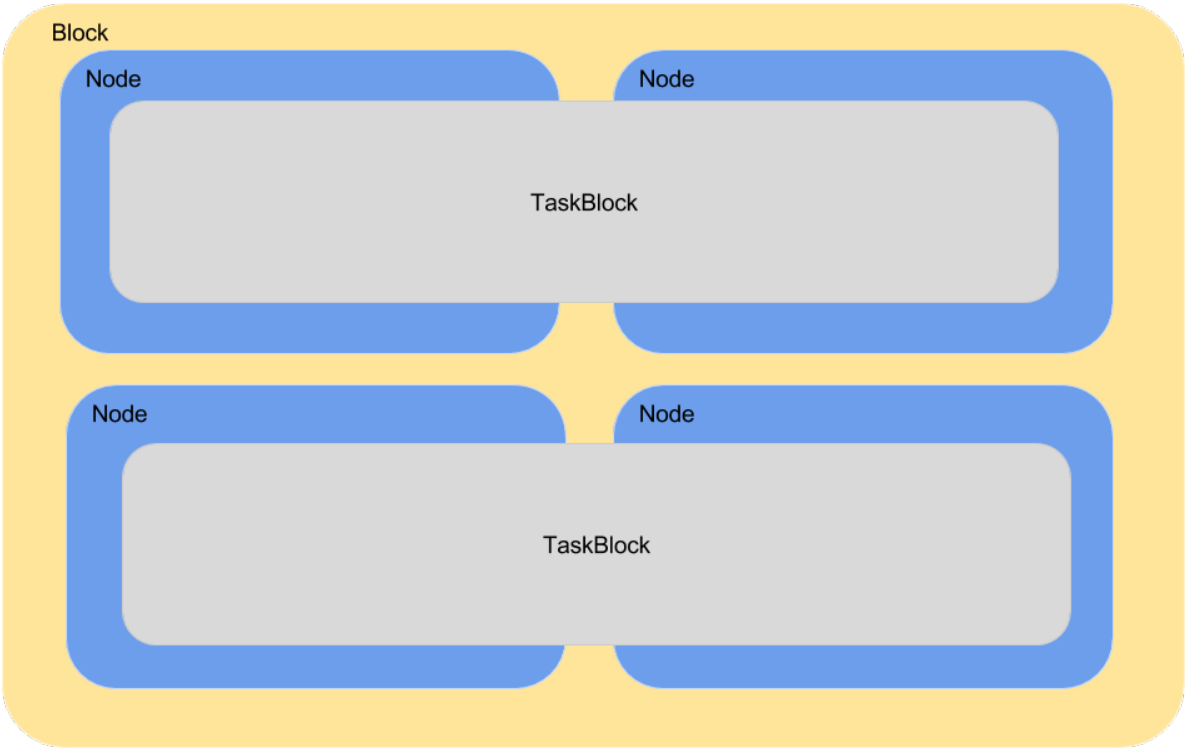
1. A single Block comprised of a Node with one TaskBlock :



2. A single Block comprised on a Node with several TaskBlocks. This configuration is most suitable for single threaded python/bash applications running on multicore target systems. With TaskBlocks proportional to the cores available on the system, apps can execute in parallel.



3. A Block comprised of several Nodes and several TaskBlocks. This configuration is generally used by MPI applications and requires support from specific MPI launchers supported by the target system (e.g., aprun, srun, mpirun, mpiexec).



3.6.4 Elasticity

Note: This feature is available from Parsl `v0.4.0`

Parsl implements a dynamic dependency graph in which the graph is extended as new tasks are enqueued and completed. As the Parsl script executes the workflow, new tasks are added to a queue for execution. Tasks are then executed asynchronously when their dependencies are met. Parsl uses the selected executor(s) to manage task execution on the execution provider(s). The execution resources, like the workflow, are not static: they can be elastically scaled to handle the variable workload generated by the workflow.

As Parsl manages a dynamic dependency graph, it does not know the full "width" of a particular workflow a priori. Further, as a workflow executes, the needs of the tasks may change, as well as the capacity available on execution providers. Thus, Parsl must elastically scale the resources it is using. To do so, Parsl includes an extensible flow control system that monitors outstanding tasks and available compute capacity. This flow control monitor, which can be extended or implemented by users, determines when to trigger scaling (in or out) events to match workflow needs.

The animated diagram below shows how blocks are elastically managed within a site. The script configuration for a site defines the minimum, maximum, and initial number of blocks to be used. Depending on workload, Parsl provisions or deprovisions blocks.

The configuration options for specifying elasticity bounds are:

1. `minBlocks`: Minimum number of blocks to maintain per site.

2. `initBlocks`: Initial number of blocks to provision at initialization of workflow.
3. `maxBlocks`: Maximum number of blocks that can be active at a site from one workflow.

Parallelism

Parsl provides a simple user-managed model for controlling elasticity. It allows users to prescribe the minimum and maximum number of blocks to be used on a given site as well as a parameter (p) to control the level of parallelism. Parallelism is expressed as the ratio of TaskBlocks to active tasks. Recall that each TaskBlock is capable of executing a single task at any given time. A parallelism value of 1 represents aggressive scaling where as many resources as possible are used; parallelism close to 0 represents the opposite situation in which as few resources as possible (i.e., minBlocks) are used.

For example:

- When p = 0: Use the fewest resources possible. Infinite tasks are stacked per TaskBlock.

```
if active_tasks == 0:
    blocks = minBlocks
else:
    blocks = max(minBlocks, 1)
```

- When p = 1: Use as many resources as possible. One task is stacked per TaskBlock.

```
blocks = min(maxBlocks,
             ceil(active_tasks / TaskBlocks))
```

- When p = 1/2: Stack up to 2 tasks per TaskBlock before overflowing and requesting a new block.

Configuration

The example below shows how elasticity and parallelism can be configured. Here, a local IPythonParallel environment is used with a minimum of 1 block and a maximum of 2 blocks, where each block may host up to 4 TaskBlocks. Parallelism of 0.5 means that when more than 2 tasks are queue per TaskBlock a new block will be requested (up to two possible blocks).

```
localIPP = {
    "sites": [
        {"site": "Local_IPP",
         "auth": {
             "channel": None,
         },
         "execution": {
             "executor": "ipp",
             "provider": "local",
             "block": {
                 "minBlocks" : 1,
                 "maxBlocks" : 2, # Shape of the blocks
                 "initBlocks": 1,
                 "TaskBlocks": 4, # Number of workers in a block
                 "parallelism" : 0.5
             }
         }
        }]
}
```

The animated diagram below illustrates the behavior of this site. In the diagram, the tasks are allocated to the first block, until 5 tasks are submitted. At this stage, as more than 2 tasks are waiting per TaskBlock, Parsl provisions a new block for executing the remaining tasks.

3.6.5 Multi-Site

Note: This feature is available from Parsl 0.4.0

Parsl supports the definition of any number of execution sites in the configuration, as well as specifying which of these sites could execute specific apps.

The common scenarios for this feature are:

- The workflow has an initial simulation stage that runs on the compute heavy nodes of an HPC system followed by an analysis and visualization stage that is better suited for the GPU nodes.
- The workflow follows a repeated fan-out, fan-in model where the long running fan-out tasks are computed on a cluster and the quick fan-in computation is better suited for execution using threads on the login node.
- The workflow includes apps that wait and evaluate the results of a computation to determine whether the app should be relaunched. Only apps running on threads may launch apps. Often, science simulations have stochastic behavior and may terminate before completion. In such cases, having a wrapper app that checks the exit code and determines whether or not the app has completed successfully can be used to automatically re-execute the app (possibly from a checkpoint) until successful completion.

Here's a code snippet that shows how sites can be specified in the `App` decorator.

```
#(CPU Heavy app)  (CPU Heavy app)  (CPU Heavy app) <--- Run on compute queue
#       |                 |                |
#    (data)            (data)           (data)
#       \                |               /
#       (Analysis & Visualization phase)         <--- Run on GPU node

# A mock Molecular Dynamics simulation app
@App('bash', dfk, sites=["Theta.Phi"])
def MD_Sim(arg, outputs=[]):
    return "MD_simulate {} -o {}".format(arg, outputs[0])

# Visualize results from the mock MD simulation app
@App('bash', dfk, sites=["Cooley.GPU"])
def Visualize(inputs=[], outputs=[]):
    bash_array = " ".join(inputs)
    return "viz {} -o {}".format(bash_array, outputs[0])
```

3.7 Error Handling

In this section we will cover the various mechanisms Parsl provides to add resiliency and robustness to workflows.

3.7.1 Exceptions

Apps fail in remote settings due to a variety of reasons. To handle errors Parsl captures, tracks, and provides functionality to appropriately respond to failures during workflow execution. A specific executing instance of an App is called a **task**. If a task is unable to complete execution within specified time limits and produce the specified set of outputs it is considered to have failed.

Failures might occur to one or more of the following reasons:

1. Task exceed specified walltime.
2. Formatting error while formatting the command-line string in `Bash Apps`
3. Task failed during execution
4. Task completed execution but failed to produce one or more of its specified outputs.
5. The App failed to launch, for example if an input dependency is not met.

Since Parsl tasks are executed asynchronously, we are faced with the issue of determining where to place exception handling code in the workflow. In Parsl all exceptions are associated with the task futures. These exceptions are raised only when a result is called on the future of a failed task. For example:

```
@App('python', dfk)
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')
```

3.7.2 Retries

Retries are one of the simplest and most frequently used methods to add resiliency to `App` failures. By retrying failed apps, transient failures (eg. machine failure, network failure) and intermittent failures within applications can be addressed. When `retries` are enabled (set to integer > 0), parsl will automatically re-launch applications that have failed, until the retry limit is reached. This feature will be available starting in Parsl *v0.5.0*.

By default `retries = 0`. Retries can be enabled by setting `retries` in the config passed to the DataFlowKernel, or as an explicit keyword argument to the DataFlowKernel at its initialization.

Here is an example of setting retries via the config:

```
from parsl import DataFlowKernel, App
from parsl.tests.configs.local_threads import config
config["globals"]["retries"] = 2

dfk = DataFlowKernel(config=config)
```

Here is an example of setting retries via keyword argument to the DFK:

```
from parsl import DataFlowKernel, App
from parsl.tests.configs.local_ipp import config

dfk = DataFlowKernel(config=config, retries=2)
```

3.7.3 Lazy Fail

While Retries address resiliency at the level of `Apps`, lazy failure adds resiliency at the workflow level. When lazy failures are enabled, the workflow does not halt as soon as it encounters a failure, but continues execution of every app that is unaffected. Lazy failures is the default behavior in parsl, with the expectation that when running production workflows, individual app failures can be deferred until the end of the workflow. During the development and testing of workflows, failing immediately on any failure is often preferred and this behavior is possible by setting `lazyErrors=False`.

For eg:

```
Here's a workflow graph, where
     (X)  is runnable,
     [X]  is completed,
     (X*) is failed.
     (!X) is dependency failed

  (A)            [A]            (A)
  / \            / \            / \
(B) (C)        [B] (C*)       [B] (C*)
 |   |    =>    |   |    =>    |   |
(D) (E)        (D) (E)        [D] (!E)
  \ /            \ /            \ /
  (F)            (F)            (!F)

  time ----->
```

There are two ways to disable lazy failures: via setting `config['globals']['lazyErrors']=False` or by setting `lazyErrors=False` as keyword argument to the DataFlowKernel at initialization.

Here is an example of disabling lazy failures via the config passed to the DFK

```
from parsl import DataFlowKernel, App
from parsl.tests.configs.local_ipp import config
config["globals"]["lazyErrors"] = False

dfk = DataFlowKernel(config=config)
```

Here is an example of disabling lazy failures via keyword argument to the DFK:

```
from parsl import DataFlowKernel, App
from parsl.tests.configs.local_ipp import config

dfk = DataFlowKernel(config=config, lazyErrors=False)
```

Note: The naming inconsistency in the config option and kwargs to dfk will be fixed in 0.5.0 release

3.8 AppCaching

When developing a workflow, developers often run the same workflow with incremental changes over and over. Often large fragments of a workflow will not have changed, yet apps will be executed again, wasting valuable developer time and computation resources. `AppCaching` solves this problem by storing results from apps that have completed so that they can be re-used. By default caching is **not** enabled. It must be explicitly enabled, either globally via the configuration, or on each app for which caching is desired.

```
@app('bash', dfk, cache=True)
def hello (msg, stdout=None):
    return 'echo {}'.format(msg)
```

AppCaching can be particularly useful when developing interactive workflows such as when using a Jupyter notebook. In this case, cells containing apps are often re-executed as during development. Using AppCaching will ensure that only modified apps are re-executed.

3.8.1 Caveats

It is important to consider several important issues when using AppCaching:

- Determinism: AppCaching is generally useful only when the apps are deterministic. If the outputs may be different for identical inputs, AppCaching will hide this non-deterministic behavior. For instance, caching an app that returns a random number will result in every invocation returning the same result.
- Timing: If several identical calls to a previously defined app are made for the first time, many instances of the app will be launched as no cached result is yet available. Once one such app completes and the result is cached all subsequent calls will return immediately with the cached result.
- Performance: If AppCaching is enabled, there is likely to be some performance overhead especially if a large number of short duration tasks are launched rapidly.

Note: The performance penalty has not yet been quantified.

3.8.2 Configuration

AppCaching may be disabled globally in the configuration. If the `appCache` is set to `False` all AppCaching is disabled. By default the global `appCache` is **enabled**; however, AppCaching for each app is disabled by default. Thus, users must explicitly enable AppCaching on each app.

AppCaching can be disabled globally in the config as follows:

```
config = {
    "sites": [{ ... }],
    "globals": {
         "appCache": False # <-- Disable AppCaching globally
}

dfk = DataFlowKernel(config=config)
```

3.9 Checkpointing

Large scale workflows are prone to errors due to node failures, application or environment errors, and myriad other issues. Parsl's support for checkpointing provides workflow resilence and fault tolerance.

Note: Checkpointing is *only* possible for apps which have AppCaching enabled. If AppCaching is disabled in `config['globals']`, checkpointing will **not** work.

Parsl follows an incremental checkpointing model, where each checkpoint contains all results that have been updated since the last checkpoint.

When loading a checkpoint the Parsl script will use checkpointed results for any apps that have been previously executed. Like AppCaching, checkpoints use the app name, hash, and input parameters to locate previously computed results. If multiple checkpoints exist for an app (with the same hash) the most recent entry will be used.

Checkpointing works in the following modes:

1. `task_exit`: In this mode, a checkpoint is created each time an app completes or fails (after retries if enabled). This mode reduces the risk of losing information from completed tasks to a minimum.

```
>>> config["globals"]["checkpointMode"] = 'task_exit'
```

2. `periodic`: The periodic mode allows the user to specify the interval at which all tasks are checkpointed.

```
>>> config["globals"]["checkpointMode"] = 'periodic'
>>> config["globals"]["checkpointPeriod"] = "01:00:00"
```

3. `dfk_exit`: In this mode, checkpoints are created when the DataFlowKernel is about to exit. This reduces the risk of losing results due to premature workflow termination from exceptions, terminate signals, etc. However there's still some likelihood that information might be lost if the workflow is terminated abruptly (machine failure, SIGKILL etc)

```
>>> config["globals"]["checkpointMode"] = 'dfk_exit'
```

4. Manual: In addition to these automated checkpointing modes, it is also possible to manually initiate a checkpoint by calling `DataFlowKernel.checkpoint()` in the workflow code.

In all cases the checkpoint is written out to the `runinfo/RUN_ID/checkpoint/` directory.

3.9.1 Creating a checkpoint

When using automated checkpointing there is no need for users to modify their Parsl script in any way: checkpointing will be conducted completely automatically. The following example shows how manual checkpointing can be invoked in a Parsl script.

```
from parsl import *
from parsl.tests.configs.local_threads import config

dfk = DataFlowKernel(config=config)

@App('python', dfk, cache=True)
def slow_double(x, sleep_dur=1):
    import time
    time.sleep(sleep_dur)
    return x * 2
```

(continues on next page)

(continued from previous page)

```
N = 5   # Number of calls to slow_double
d = []  # List to store the futures
for i in range(0, N):
    d.append(slow_double(i))

# Wait for the results
[i.result() for i in d]

cpt_dir = dfk.checkpoint()
print(cpt_dir)  # Prints the checkpoint dir
```

3.9.2 Loading a checkpoint

To load a checkpoint the user must select which checkpoint file to resume from. As mentioned above, checkpoint files are stored in the `runinfo/RUNID/checkpoint` directory. The example below shows how to resume using from all available checkpoints:

```
from parsl import *
from parsl.tests.configs.local_threads import config
from parsl.utils import get_all_checkpoints

dfk = DataFlowKernel(config=config,
                     checkpointFiles=parsl.get_all_checkpoints())
```

3.10 Configuration

Parsl workflows are developed completely independently from their execution environment. Parsl offers an extensible configuration model through which the execution environment and communication with that environment is configured.

Parsl can be configured using a Python configuration object. For simple cases, such as threads these configurations can be easily specified inline. For more complex environments using different block configurations and communication channels it is easiest to define a full configuration object. The following shows how the configuration can be passed to the Dataflow Kernel.

1. **Executors**, which use threads, iPyParallel workers, etc. can be constructed manually

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

2. A **config** can 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,
   ```

(continues on next page)

(continued from previous page)

```
                "maxThreads" : 4
            }
        }],
    "globals" : {"lazyErrors" : True}
}
dfk = DataFlowKernel(config=config)
```

3.10.1 Configuration Structure

The configuration data structure is a python dictionary that describes execution sites as well as other information such as global attributes, controller information, and in the future data staging information.

```
{
  "sites" : [ list of site definitions ],
  "globals" : { dict of attributes global to the workflow }
  "controller" : { dict of attributes specific to the local IPP controller(s) }
}
```

The most important part of this configuration is the *sites* key. Parsl allows multiple sites to be specified in a list. The configuration for an individual site is as follows:

```
{
   "site" : < str name of the site being defined>

   # dictionary of attributes that define how the execution resource is accessed
   "auth" : {
      # Define the channel type used to reach the site
      "channel" : <str (local, ssh, ssh-il)>,
   }

   # The execution block defines how resources can be requested and how the resources␣
↪should be
   # shaped to best match the workflow needs.
   "execution" : {
       # The executor is the mechanism that executes tasks on the compute
       # resources provisioned from the site
       "executor" : <str (ipp, threads, swift_t)>,

       # Select the kind of scheduler or resource type of the site
       "provider" : <str (slurm, torque, cobalt, condor, aws, azure, local ...)>

       # A block is the unit by which resources are requested from the site
       "block" : {
            "nodes"      : <int: nodes to request per block>,
            "taskBlocks" : <str: workers to start per block, or bash expression>,
            "initBlocks" : <int: blocks to provision at the execution start>,
            "minBlocks"  : <int: min blocks to maintain during execution>,
            "maxBlocks"  : <int: max blocks that can be provisioned>,
            "walltime"   : <str: walltime allowed for the block in HH:MM:SS format>,

            # The "options" block contains attributes that are provider specific
            # such as scheduler options
            "options" : {
                 #dict of provider specific attributes, please refer to provider
                 # specific documentation.
```

(continues on next page)

(continued from previous page)

```
            }
        }
    }
}
```

The following shows an example configuration for accessing NERSC's Cori supercomputer. This example uses the IPythonParallel executor and connects to Cori's Slurm scheduler. It uses a remote SSH channel that allows the IPythonParallel controller to be hosted on the script's submission machine (e.g., a PC). It is configured to request 2 nodes configured with 1 TaskBlock per node. Finally it includes override information to request a particular node type (Haswell) and to configure a specific Python environment on the worker nodes using Anaconda.

```
config = {
    "sites" : [
        { "site" : "Cori.Remote.IPP",
          "auth" : {
              "channel" : "ssh",
              "hostname" : "cori.nersc.gov",
              "username" : "username",
              "scriptDir" : "/global/homes/y/username/parsl_scripts"
          },
          "execution" : {
              "executor" : "ipp",
              "provider" : "slurm",
              "block" : {
                  "nodes" : 2,
                  "taskBlocks" : 1,
                  "walltime" : "00:10:00",
                  "initBlocks" : 1,
                  "minBlocks" : 0,
                  "maxBlocks" : 1,
                  "scriptDir" : ".",
                  "options" : {
                      "partition" : "debug",
                      "overrides" : """#SBATCH --constraint=haswell
module load python/3.5-anaconda ;
source activate /global/homes/y/yadunand/.conda/envs/parsl_env_3.5"""
                  }
              }
          }
        }
        ],
    "globals" : {   "lazyErrors" : True },
}
```

3.11 Importing Parsl apps

It may be convenient to define Parsl apps separately from the definition of the `DataFlowKernel`, or in libraries of apps which are intended to be imported by other modules. For this reason, the *`DataFlowKernel`* is an optional argument to the *`App()`* decorator. If the `DataFlowKernel` is not passed to the *`App()`* decorator, a configuration must be loaded using `parsl.load` prior to calling the app.

The configuration can be defined in the Parsl script, or elsewhere before being imported. As an example of the latter, consider a file called `config.py` which contains the following definition:

```
local_threads = {
    "sites": [
        {
            "site": "local_threads",
            "auth": {
                "channel": None
            },
            "execution": {
                "executor": "threads",
                "provider": None,
                "maxThreads": 4
            }
        }
    ],
    "globals": {
        "lazyErrors": True
    }
}
```

In a separate file called `library.py`, we define:

```
from parsl import App

@App('python')
def increment(x):
    return x + 1
```

Putting these together in a third file called `run_increment.py`, we load the configuration from `config.py` before calling the `increment` app:

```
import parsl
from config import local_threads
from library import increment

parsl.load(local_threads)

for i in range(5):
    print('{} + 1 = {}'.format(i, increment(i).result()))
```

Which produces the following output:

```
0 + 1 = 1
1 + 1 = 2
2 + 1 = 3
3 + 1 = 4
4 + 1 = 5
```

3.12 Usage Statistics Collection

Parsl sends usage statistics back to the Parsl development team to measure worldwide usage and and improve reliability and usability.

3.12.1 Why are we doing this?

The Parsl development team receives support from government funding agencies. For the team to continue to receive such funding, and for the agencies themselves to argue for funding, both the team and the agencies must be able to demonstrate that the scientific community is benefiting from these investments. To this end, we want to provide generic usage data about such things as the following:

- How many people use Parsl
- Average job length
- Parsl exit codes

To this end, we have added support to Parsl that allows installations to send us generic usage statistics. By participating in this project, you help justify continuing support for the software on which you rely. The data sent is as generic as possible (see What is sent? below).

3.12.2 Opt-Out

We have chosen opt-out collection rather than opt-in. The reason is that we need this data - it is a requirement for funding. We believe have set a good balance between the benefits to the project and the users by showing that Parsl works and is in use, which helps the project continue, and the costs to users of providing generic information. To keep the cost low, we need to require zero additional effort. By not opting out, and allowing these statistics to be reported back, you are explicitly supporting the further development of Parsl.

If you must opt out of usage reporting, set `PARSL_TRACKING=false` in your environment.

3.12.3 What is sent?

- Anonymized user ID
- Anonymized hostname
- Anonymized Parsl script ID
- Start and end times
- Parsl exit code
- Count of sites used

3.12.4 How is the data sent?

The data is sent via UDP. While this may cause us to lose some data, it drastically reduces the possibility that the usage statistics reporting will adversely affect the operation of the software.

3.12.5 When is the data sent?

The data is sent twice per run, once when Parsl starts a script, and once when the script is completed.

3.12.6 What will the data be used for?

The data will be used for answering questions such as:

- How many unique users are using Parsl?

- To determine patterns of usage - is activity increasing or decreasing?

We will also try and mine the data to answer operational questions such as:

- What percentage of the jobs run complete successfully?
- Of the ones that fail, what is the most common fault code returned?

3.12.7 Feedback

Please send us your feedback at parsl@googlegroups.com. Feedback from our user communities will be useful in determining our path forward with this in the future. We do ask that if you have concerns or objections, please be specific in your feedback. For example, while saying "Our site has a policy against sending such data" is good information for us to know in the future, a link to such a policy would be even better.

3.13 Container Support

There are two broad models for app execution with containers:

1. Workers are launched inside containers; a single container can be re-used for several `Apps`.
2. Each `App` is launched inside a fresh container.

This document describes the first case. In this model, the `Apps` are executed on a worker that is launched within a container. For simplicity we focus on Docker although the same methods can be extended to supported other container systems such as Singularity, Shifter etc.

Caution: This feature is available from Parsl `v0.5.0` in an `experimental` state. We request feedback and feature enhancement requests via github.

3.13.1 Docker

The following section describes creating a pool of containers, each with a worker that executes specific `Apps`. Most of the immediately following sections can be skimmed if you have experience working with containers.

Installing Docker

To install Docker please ensure you have sudo privileges and follow instructions here.

Once installed make sure that Docker is installed:

```
# Get the Docker version
docker --version

# Get Docker info/stats
docker info

# Do a quick check with hello-world
docker run hello-world
```

Creating an Image

Please note that the following instructions are tested on Ubuntu 16.04. If you are on a different operating system, every command that is not a Docker command might need to be tweaked for your specific system. Such cases will be noted explicitly.

1. Pull an image with the latest python.

```
# Get a basic python image
docker pull python
```

2. Construct a new python image with your modifications by creating a file called `Dockerfile` with the following contents. Every command in the container definition is assumed to be running in Ubuntu.

```
# Use an official Python runtime as a parent image
FROM python:3.6

# Set the working directory to /home
WORKDIR /home

# Install any needed packages specified in requirements.txt
RUN pip3 install parsl
```

3. Once your updates are made, create a Docker image from the Dockerfile.

```
docker build -t parslbase_v0.1 .
```

4. Make sure your user has privileges to launch and manage Docker by adding yourself to the `docker` group. The following command assumes an Ubuntu machine.

```
sudo usermod -a -G docker $USER
```

5. Ensure that you are running `Python3.6.X`. If you need python3.5, make sure that the container built in the previous steps install and setup the python version that match the host machine's environment.

```
# This command should return Python 3.6 or higher.
python3 -V
```

6. Set up apps. Check the following directories for two simple apps:

   - `parsl/docker/app1`
   - `parsl/docker/app2`

   These container scripts are setup such that, when they are built they copy the application python code over to `/home`, which will be the `cwd` when app invocations are made. Each of these *appN.py* scripts contain the definition of a `predict(List)` function.

7. Build the test applications as Docker images: We assume you are in the top level of the Parsl repository.

```
# Docker build app1
cd docker/app1
docker build -t app1_v0.1 .

# Docker build the next app
cd ../app2
docker build -t app2_v0.1 .
```

(continues on next page)

(continued from previous page)

```
# Check the new images:
docker images list
```

Parsl Config

Now that we have a Docker image available locally, we will create a `site` that uses such an image to launch containers. `Apps` will execute in this environment.

Here is a Parsl configuration using one of the Docker images created in the previous section.

```
local_docker_IPP = {
    "sites": [
        {
            "site": "pool_app1",
            "auth": {
                "channel": None
            },
            "execution": {
                "executor": "ipp",
                "container": {
                    "type": "docker",  # Specify Docker
                    "image": "app1_v0.1",  # Specify docker image
                },
                "provider": "local",
                "block": {
                    "initBlocks": 2,  # Start with 4 workers
                },
            }
        }
    ],
    "globals": {
        "lazyErrors": True
    }
}
```

For workflows with multiple apps which require different docker images, a new site should be created for each of the images that will be used. In the Parsl workflow definition the `App` decorator can then be tagged with the `sites` keyword argument to ensure that apps execute on the specific sites with the right container image.

Caution: If you have specific modules or python packages that are imported from relative paths, the workers in the container will not have these available unless explicitly copied in.

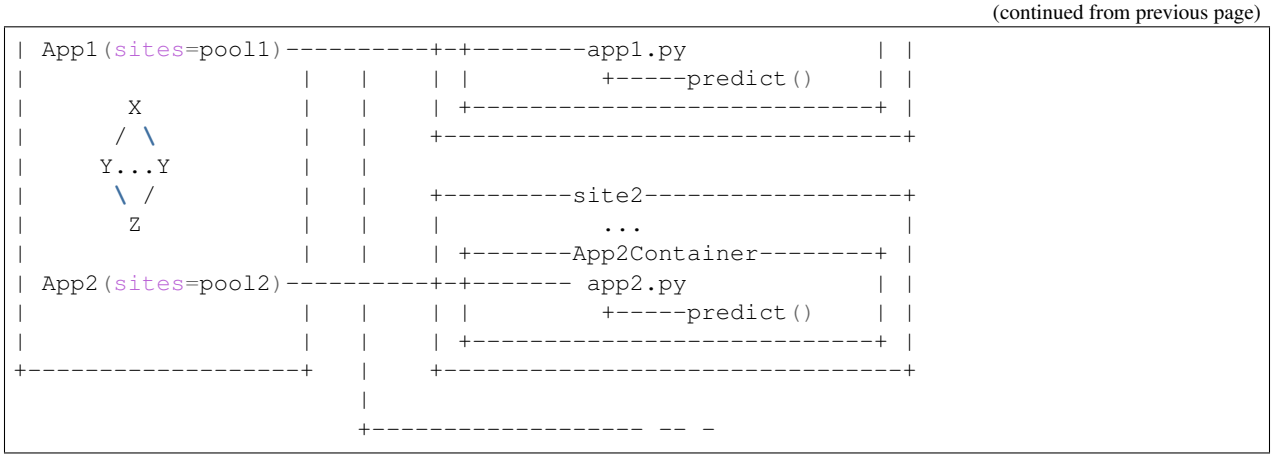
```
$ DOCKER_CWD=$(docker image inspect --format='{{{{.Config.WorkingDir}}}}' {2})
$ docker cp -a . $DOCKER_ID:$DOCKER_CWD
```

How this works

```
                        +-----local/Kubernetes/slurm... ---
                        |
+----- Parsl--------+   |    +---------site1------------------+
|                    |   |    |           ...                 |
|                    |   |    | +-------App1Container--------+ |
```

(continues on next page)

(continued from previous page)



The diagram above illustrates the various components and how the interact with each other to act as a fast model serving system. In this model, each site in the Parsl config definition can only serve one container image. Parsl launches multiple blocks matching the definition of the site, and each block will contain one container instantiated with a worker running inside. In the examples given above, the worker is launched in the working directory which also contains some application code:`app1.py`.

The application codes `app1.py` and `app2.py` in our example Docker images, both contain a simple python function `predict()` that takes a list of numbers (floats/ints) applies a simple arithmetic operation and returns a corresponding list.

Here's the contents of `app1.py`:

```
def predict(list_items):
    """Returns the double of the items"""
    return [i*2 for i in list_items]
```

A snippet of the Parsl code that imports the `app1.py` file and calls `predict()` on a site that specifies the right container image `app1_v0.1` is below :

```
@App('python', dfk, sites=['pool_app1'], cache=True)
def app_1(data):
    import app1
    return app1.predict(data)

x = app_1([1,2,3])

# The print statement prints [2,4,6] once the results are available
print(x.result())
```

CHAPTER 4

FAQ

4.1 How can I debug a Parsl script?

Parsl interfaces with the Python logger. To enable logging of parsl's progress to stdout, turn on the logger as follows. Alternatively, you can configure the file logger to write to an output file.

```
from parsl import *
import logging

# Emit log lines to the screen
parsl.set_stream_logger()

# Write log to file, specify level of detail for logs
parsl.set_file_logger(FILENAME, level=logging.DEBUG)
```

Note: Parsl's logging will not capture STDOUT/STDERR from the apps themselves. Follow instructions below for application logs.

4.2 How can I view outputs and errors from Apps?

Parsl Apps include keyword arguments for capturing stderr and stdout in files.

```
@app('bash', dfk)
def hello (msg, stdout=None):
    return 'echo {}'.format(msg)

# When hello() runs the STDOUT will be written to 'hello.txt'
hello('Hello world', stdout='hello.txt')
```

4.3 How can I make an App dependent on multiple inputs?

You can pass any number of futures in to a single App either as positional arguments or as a list of futures via the special keyword *inputs=[]*. The App will wait for all inputs to be satisfied before execution.

4.4 Can I pass any Python object between Apps?

No. Unfortunately, only picklable objects can be passed between Apps. For objects that can't be pickled, it is recommended to use object specific methods to write the object into a file and use files to communicate between Apps.

4.5 How do I specify where Apps should be run?

Parsl's multi-site support allows you to define the site (including local threads) on which an App should be executed. For example:

```
@app('python', dfk, sites=['SuperComputer1'])
def BigSimulation(...):
    ...

@app('python', dfk, sites=['GPUMachine'])
def Visualize (...)
    ...
```

4.6 Workers do not connect back to Parsl

If you are running via ssh to a remote system from your local machine, or from the login node of a cluster/supercomputer, it is necessary to have a public IP to which the workers can connect back. While our pilot job system, ipyparallel, can identify the IP address automatically on certain systems, it is safer to specify the address explicitly.

Here's how you specify the address in the config dictionary passed to the DataFlowKernel:

```
multiNode = {
    "sites": [{
        "site": "ALCF_Theta_Local",
        "auth": {
            "channel": "ssh",
            "scriptDir": "/home/{}/parsl_scripts/".format(USERNAME)
        },
        "execution": {
            "executor": "ipp",
            "provider": '<SCHEDULER>'
            "block": { # Define the block
                ...
            }
        },
    }],
    "globals": {
        "lazyErrors": True,
},
```

(continues on next page)

(continued from previous page)

```
    "controller": {
    "publicIp": '<AA.BB.CC.DD>'   # <--- SPECIFY PUBLIC IP HERE
    }
}
```

4.7 Remote execution fails with SystemError(unknown opcode)

When running with Ipyparallel workers, it is important to ensure that the Python version on the client side matches that on the side of the workers. If there's a mismatch, the apps sent to the workers will fail with the following error: `ipyparallel.error.RemoteError: SystemError(unknown opcode)`

Caution: It is **required** that both the parsl script and all workers are set to use python with the same Major.Minor version numbers. For example, use Python3.5.X on both local and worker side.

4.8 Parsl complains about missing packages

If `parsl` is cloned from a github repository and added to the `PYTHONPATH`, it is possible to miss the installation of some dependent libraries. In this configuration, `parsl` will raise errors such as:

```
ModuleNotFoundError: No module named 'ipyparallel'
```

In this situation, please install the required packages. If you are on a machine with sudo privileges you could install the packages for all users, or if you choose, install to a virtual environment using packages such as virtualenv and conda.

For instance, with conda, follow this cheatsheet to create a virtual environment:

```
# Activate an environmentconda install
source active <my_env>

# Install packages:
conda install <ipyparallel, dill, boto3...>
```

4.9 zmq.error.ZMQError: Invalid argument

If you are making the transition from Parsl v0.3.0 to v0.4.0 and you run into this error, please check your config structure. In v0.3.0, `config['controller']['publicIp'] = '*'` was commonly used to specify that the IP address should be autodetected. This has changed in v0.4.0 and setting `'publicIp' = '*'` results in an error with a traceback that looks like this:

```
File "/usr/local/lib/python3.5/dist-packages/ipyparallel/client/client.py", line 483,␣
↪in __init__
self._query_socket.connect(cfg['registration'])
File "zmq/backend/cython/socket.pyx", line 528, in zmq.backend.cython.socket.Socket.
↪connect (zmq/backend/cython/socket.c:5971)
File "zmq/backend/cython/checkrc.pxd", line 25, in zmq.backend.cython.checkrc._check_
↪rc (zmq/backend/cython/socket.c:10014)
zmq.error.ZMQError: Invalid argument
```

In v0.4.0, the controller block defaults to detecting the IP address automatically, and if that does not work for you, you can specify the IP address explicitly like this: `config['controller']['publicIp'] = 'IP.ADD.RES.S'`

4.10 How do I run code that uses Python2.X?

Modules or code that require Python2.X cannot be run as python apps, however they may be run via bash apps. The primary limitation with python apps is that all the inputs and outputs including the function would be mangled when being transmitted between python interpreters with different version numbers (also see *Remote execution fails with SystemError(unknown opcode)*)

Here's an example of running a python2.7 code as a bash application:

```
@app('bash', dfk)
def python_27_app (arg1, arg2 ...):
    return '''conda activate py2.7_env  # Use conda to ensure right env
    python2.7 my_python_app.py -arg {0} -d {1}
    '''.format(arg1, arg2)
```

4.11 Parsl hangs

There are a few common situations in which a Parsl script might hang:

1. Circular Dependency in code If an *app* takes a list as an *input* argument and the future returned is added to that list, it creates a circular dependency that cannot be resolved. This situation is described here in more detail.
2. Workers requested are unable to contact the Parsl client due to one or more issues listed below:
   - Parsl client does not have a public IP (e.g. laptop on wifi). If your network does not provide public IPs, the simple solution is to ssh over to a machine that is public facing. Machines provisioned from cloud-vendors setup with public IPs are another option.
   - Parsl hasn't autodetected the public IP. This can be resolved by manually specifying the public IP via the config:

     ```
     config["controller"]["publicIp"] = 8.8.8.8
     ```

   - Firewall restrictions that block certain port ranges. If there is a certain port range that is **not** blocked, you may specify that via the config:

     ```
     # Assuming ports 50000 to 55000 are open
     config["controller"]["portRange"] = "50000,55000"
     ```

4.12 How can I start a Jupyter notebook over SSH?

See instructions here.

4.13 How can I sync my conda environment and Jupyter environment?

Run:

```
conda install nb_conda
```

Now all available conda environments (for example, one created by following the instructions *here*) will automatically be added to the list of kernels.

CHAPTER 5

Reference guide

parsl.set_stream_logger	Add a stream log handler.
parsl.set_file_logger	Add a stream log handler.
parsl.app.app.App	The App decorator function.
parsl.app.futures.DataFuture	A datafuture points at an AppFuture.
parsl.dataflow.futures.AppFuture	An AppFuture points at a Future returned from an Executor.
parsl.dataflow.dflow.DataFlowKernelLoader	Manage which DataFlowKernel is active.
parsl.data_provider.files.File	The Parsl File Class.

5.1 parsl.set_stream_logger

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

5.2 parsl.set_file_logger

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

5.3 parsl.app.app.App

parsl.app.app.**App**(*apptype*, *executor=None*, *walltime=60*, *cache=False*, *sites='all'*)
The App decorator function.

Args:

- apptype (string) : Apptype can be bash|python

Kwargs:

- executor (Executor): Executor for the execution resource. This can be omitted only after calling `parsl.dataflow.dflow.DataFlowKernelLoader.load()`.
- **walltime (int)** [Walltime for app in seconds,] default=60
- **sites (str|List)** [List of site names on which the app could execute] default='all'
- **cache (Bool)** [Enable caching of the app call] default=False

Returns: An AppFactory object, which when called runs the apps through the executor.

5.4 parsl.app.futures.DataFuture

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.

__init__(*fut*, *file_obj*, *parent=None*, *tid=None*)
Construct the DataFuture object.

If the file_obj is a string convert to a File.

Args:

- fut (AppFuture) : AppFuture that this DataFuture will track
- file_obj (string/File obj) : Something representing file(s)

Kwargs:

- parent ()
- tid (task_id) : Task id that this DataFuture tracks

Methods

`__init__`(fut, file_obj[, parent, tid])	Construct the DataFuture object.
`add_done_callback`(fn)	Attaches a callable that will be called when the future finishes.
`cancel`()	Cancel the task that this DataFuture is tracking.
`cancelled`()	Return True if the future was cancelled.
`done`()	Return True of the future was cancelled or finished executing.
`exception`([timeout])	Return the exception raised by the call that the future represents.
`parent_callback`(parent_fu)	Callback from executor future to update the parent.
`result`([timeout])	A blocking call that returns either the result or raises an exception.
`running`()	Return True if the future is currently executing.
`set_exception`(exception)	Sets the result of the future as being the given exception.
`set_result`(result)	Sets the return value of work associated with the future.
`set_running_or_notify_cancel`()	Mark the future as running or process any cancel notifications.

Attributes

`filename`	Filepath of the File object this datafuture represents.
`filepath`	Filepath of the File object this datafuture represents.
`tid`	Returns the task_id of the task that will resolve this DataFuture.

5.5 parsl.dataflow.futures.AppFuture

class parsl.dataflow.futures.**AppFuture**(*parent*, *tid=None*, *stdout=None*, *stderr=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.

__init__(*parent*, *tid=None*, *stdout=None*, *stderr=None*)

Initialize the AppFuture.

Args:

- parent (Future) : The parent future if one exists A default value of None should be passed in if app is not launched

KWargs:

- tid (Int) : Task id should be any unique identifier. Now Int.
- **stdout (str)** [Stdout file of the app.] Default: None
- **stderr (str)** [Stderr file of the app.] Default: None

Methods

`__init__`(parent[, tid, stdout, stderr])	Initialize the AppFuture.
`add_done_callback`(fn)	Attaches a callable that will be called when the future finishes.
`cancel`()	Cancel the future if possible.
`cancelled`()	Return True if the future was cancelled.
`done`()	Check if the future is done.
`exception`([timeout])	Return the exception raised by the call that the future represents.
`parent_callback`(executor_fu)	Callback from executor future to update the parent.
`result`([timeout])	Result.
`running`()	Return True if the future is currently executing.
`set_exception`(exception)	Sets the result of the future as being the given exception.
`set_result`(result)	Sets the return value of work associated with the future.
`set_running_or_notify_cancel`()	Mark the future as running or process any cancel notifications.
`update_parent`(fut)	Add a callback to the parent to update the state.

Attributes

`outputs`
`stderr`
`stdout`
`tid`

5.6 parsl.dataflow.dflow.DataFlowKernelLoader

class parsl.dataflow.dflow.**DataFlowKernelLoader**

Manage which DataFlowKernel is active.

This is a singleton class containing only class methods. You should not need to instantiate this class.

__init__()

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

Methods

`dfk`()	Return the currently-loaded DataFlowKernel.
`load`(config)	Load a DataFlowKernel.
`DataFlowKernelLoader.set_default`	

5.7 parsl.data_provider.files.File

class parsl.data_provider.files.**File**(*url*, *dman=None*, *cache=False*, *caching_dir='.'*, *staging='direct'*)

The Parsl File Class.

This is planned to be a very simple class that simply captures various attributes of a file, and relies on client-side and worker-side systems to enable to appropriate transfer of files.

__init__(*url*, *dman=None*, *cache=False*, *caching_dir='.'*, *staging='direct'*)

Construct a File object from a url string.

Args:

- **url (string)** [url string of the file e.g.]
  - 'input.txt'
  - 'file:///scratch/proj101/input.txt'
  - 'globus://go#ep1/~/data/input.txt'
  - 'globus://ddb59aef-6d04-11e5-ba46-22000b92c6ec/home/johndoe/data/input.txt'
- dman (DataManager) : data manager

Methods

__init__(url[, dman, cache, caching_dir, ...])	Construct a File object from a url string.
get_data_future(site_name)	
set_data_future(df[, site_name])	
stage_in(site_name)	Transport file from the site of origin to local site.
stage_out()	Transport file from local filesystem to origin site.

Attributes

filepath	Return the resolved filepath on the side where it is called from.

parsl.app.errors.AppBadFormatting	An error raised during formatting of a bash function.
parsl.app.errors.AppException	An error raised during execution of an app.
parsl.app.errors.AppFailure	An error raised during execution of an app.
parsl.app.errors.AppTimeout	An error raised during execution of an app when it exceeds its allotted walltime.
parsl.app.errors.BadStdStreamFile	Error raised due to bad filepaths specified for STDOUT/ STDERR.
parsl.app.errors.BashAppNoReturn	Bash app returned no string.
parsl.app.errors.DependencyError	Error raised at the end of app execution due to missing output files.
parsl.app.errors.InvalidAppTypeError	An invalid app type was requested from the @App decorator.

Continued on next page

Table 9 – continued from previous page

parsl.app.errors.MissingOutputs	Error raised at the end of app execution due to missing output files.
parsl.app.errors.NotFutureError	A non future item was passed to a function that expected a future.
parsl.app.errors.ParslError	Base class for all exceptions.
parsl.executors.errors.ControllerErr	Error raise by IPP controller.
parsl.executors.errors.ExecutorError	Base class for all exceptions.
parsl.executors.errors.ScalingFailed	Scaling failed due to error in Execution provider.
parsl.executors.exceptions.ExecutorException	Base class for all exceptions.
parsl.executors.exceptions.TaskExecException	Task execution raised an error in the remote process.

5.8 parsl.app.errors.AppBadFormatting

exception 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)

5.9 parsl.app.errors.AppException

exception parsl.app.errors.**AppException**

An error raised during execution of an app.

What this exception contains depends entirely on context

5.10 parsl.app.errors.AppFailure

exception 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)

5.11 parsl.app.errors.AppTimeout

exception parsl.app.errors.**AppTimeout**(*reason*, *exitcode*, *retries=None*)

An error raised during execution of an app when it exceeds its allotted walltime.

Contains: reason(string) exitcode(int) retries(int/None)

5.12 parsl.app.errors.BadStdStreamFile

exception parsl.app.errors.**BadStdStreamFile**(*outputs*, *exception*)

Error raised due to bad filepaths specified for STDOUT/ STDERR.

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

5.13 parsl.app.errors.BashAppNoReturn

exception parsl.app.errors.**BashAppNoReturn**(*reason*, *exitcode*, *retries=None*)
Bash app returned no string.

Contains: reason(string) exitcode(int) retries(int/None)

5.14 parsl.app.errors.DependencyError

exception 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..)

5.15 parsl.app.errors.InvalidAppTypeError

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

5.16 parsl.app.errors.MissingOutputs

exception 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..)

5.17 parsl.app.errors.NotFutureError

exception parsl.app.errors.**NotFutureError**
A non future item was passed to a function that expected a future.

This is basically a type error.

5.18 parsl.app.errors.ParslError

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

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

5.19 parsl.executors.errors.ControllerErr

exception parsl.executors.errors.**ControllerErr**(*reason*)
Error raise by IPP controller.

5.20 parsl.executors.errors.ExecutorError

exception parsl.executors.errors.**ExecutorError**
Base class for all exceptions.

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

5.21 parsl.executors.errors.ScalingFailed

exception parsl.executors.errors.**ScalingFailed**(*sitename*, *reason*)
Scaling failed due to error in Execution provider.

5.22 parsl.executors.exceptions.ExecutorException

exception parsl.executors.exceptions.**ExecutorException**
Base class for all exceptions.

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

5.23 parsl.executors.exceptions.TaskExecException

exception parsl.executors.exceptions.**TaskExecException**
Task execution raised an error in the remote process.

CHAPTER 6

Developer documentation

6.1 Contributing

Parsl is an open source project that welcomes contributions from the community.

Contributions may take many forms from reporting issues, requesting new features commenting on existing issues, fixing bugs, or developing new features.

If you're interested in contributing, please review our contributing guide.

If you're looking for a good place to get started you might like to review existing Git issues (those marked with help wanted are a good place to start).

To get involved in community discussion please join the Parsl Slack channel.

6.2 Changelog

6.2.1 Parsl 0.5.1

Released. May 15th, 2018.

New functionality

- Better code state description in logging issue#242
- String like behavior for Files issue#174
- Globus path mapping in config issue#165

Bug Fixes

- Usage tracking with certain missing network causes 20s startup delay. issue#220
- Checkpoints will not reload from a run that was Ctrl-C'ed issue#232
- Checkpoints will not reload from a run that was Ctrl-C'ed issue#220
- Race condition in task checkpointing issue#234
- *task_exit* checkpointing repeatedly truncates checkpoint file during run issue#230
- Make *dfk.cleanup()* not cause kernel to restart with Jupyter on Mac issue#212
- Fix automatic IPP controller creation on OS X issue#206
- Passing Files breaks over IPP issue#200
- *repr* call after *AppException* instantiation raises *AttributeError* issue#197
- Allow *DataFuture* to be initialized with a *str* file object issue#185
- Error for globus transfer failure issue#162

6.2.2 Parsl 0.5.0

Released. Apr 16th, 2018.

New functionality

- Support for Globus file transfers issue#71

> **Caution:** This feature is available from Parsl `v0.5.0` in an `experimental` state.

- **PathLike behavior for Files issue#174**
  - Files behave like strings here :

```
myfile = File("hello.txt")
f = open(myfile, 'r')
```

- Automatic checkpointing modes issue#106

```
config = {
    "globals": {
        "lazyErrors": True,
        "memoize": True,
        "checkpointMode": "dfk_exit"
    }
}
```

- Support for containers with docker issue#45

```
    localDockerIPP = {
         "sites": [
             {"site": "Local_IPP",
              "auth": {"channel": None},
```

(continues on next page)

(continued from previous page)

```
                "execution": {
                    "executor": "ipp",
                    "container": {
                        "type": "docker",      # <----- Specify Docker
                        "image": "app1_v0.1", # <------Specify docker image
                    },
                    "provider": "local",
                    "block": {
                        "initBlocks": 2,  # Start with 4 workers
                    },
                }
                }],
            "globals": {"lazyErrors": True}        }

.. caution::
  This feature is available from Parsl ``v0.5.0`` in an ``experimental`` state.
```

- **Cleaner logging** issue#85
  - Logs are now written by default to `runinfo/RUN_ID/parsl.log`.
  - `INFO` log lines are more readable and compact
- Local configs are now packaged issue#96

```
from parsl.configs.local import localThreads
from parsl.configs.local import localIPP
```

Bug Fixes

- Passing Files over IPP broken issue#200
- Fix *DataFuture.__repr__* for default instantiation issue#164
- Results added to appCache before retries exhausted issue#130
- Missing documentation added for Multisite and Error handling issue#116
- TypeError raised when a bad stdout/stderr path is provided. issue#104
- Race condition in DFK issue#102
- Cobalt provider broken on Cooley.alfc issue#101
- No blocks provisioned if parallelism/blocks = 0 issue#97
- Checkpoint restart assumes rundir issue#95
- Logger continues after cleanup is called issue#93

6.2.3 Parsl 0.4.1

Released. Feb 23rd, 2018.

New functionality

- GoogleCloud provider support via libsubmit

- GridEngine provider support via libsubmit

Bug Fixes

- Cobalt provider issues with job state issue#101
- Parsl updates config inadvertently issue#98
- No blocks provisioned if parallelism/blocks = 0 issue#97
- Checkpoint restart assumes rundir bug issue#95
- Logger continues after cleanup called enhancement issue#93
- Error checkpointing when no cache enabled issue#92
- Several fixes to libsubmit.

6.2.4 Parsl 0.4.0

Here are the major changes included in the Parsl 0.4.0 release.

New functionality

- Elastic scaling in response to workflow pressure. issue#46 Options *minBlocks*, *maxBlocks*, and *parallelism* now work and controls workflow execution.

  Documented in: *Elasticity*

- Multisite support, enables targetting apps within a single workflow to different sites issue#48

```
@App('python', dfk, sites=['SITE1', 'SITE2'])
def my_app(...):
    ...
```

- Anonymized usage tracking added. issue#34

  Documented in: *Usage Statistics Collection*

- AppCaching and Checkpointing issue#43

```
# Set cache=True to enable appCaching
@App('python', dfk, cache=True)
def my_app(...):
    ...


# To checkpoint a workflow:
dfk.checkpoint()
```

  Documented in: *Checkpointing*, *AppCaching*

- Parsl now creates a new directory under *./runinfo/* with an incrementing number per workflow invocation
- Troubleshooting guide and more documentation
- PEP8 conformance tests added to travis testing issue#72

Bug Fixes

- Missing documentation from libsubmit was added back issue#41
- **Fixes for *script_dir* | *scriptDir* inconsistencies issue#64**
  - We now use *scriptDir* exclusively.
- Fix for caching not working on jupyter notebooks issue#90
- Config defaults module failure when part of the option set is provided issue#74
- Fixes for network errors with usage_tracking issue#70
- PEP8 conformance of code and tests with limited exclusions issue#72
- Doc bug in recommending *max_workers* instead of *maxThreads* issue#73

6.2.5 Parsl 0.3.1

This is a point release with mostly minor features and several bug fixes

- Fixes for remote side handling
- Support for specifying IPythonDir for IPP controllers
- Several tests added that test provider launcher functionality from libsubmit
- This upgrade will also push the libsubmit requirement from 0.2.4 -> 0.2.5.

Several critical fixes from libsubmit are brought in:

- Several fixes and improvements to Condor from @annawoodard.
- Support for Torque scheduler
- Provider script output paths are fixed
- Increased walltimes to deal with slow scheduler system
- Srun launcher for slurm systems
- **SSH channels now support file_pull() method** While files are not automatically staged, the channels provide support for bi-directional file transport.

6.2.6 Parsl 0.3.0

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

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.

Bug fixes

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

6.2.7 Parsl 0.2.0

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

New functionality

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

6.2.8 Parsl 0.1.0

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

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.

Bug Fixes

- Initial release, no listed bugs.

6.3 Design and Rationale

6.3.1 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 substantiated.

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.

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.

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.

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 :

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

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

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		

6.4 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, with the release in which they were added marked for releases since 0.3.0. Help in providing any of the yet-to-be-developed capabilities is welcome.

The upcoming release is Parsl-0.6.0 and features in preparation are documented via Github issues and milestones.

6.4.1 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).**

6.4.2 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(v0.5.0)**).
- 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.

6.4.3 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 (v0.4.0)**
- **Retry logic to support recovery and fault tolerance**
  - **This can be done by the user now with Exception handling**
- **Workflow level checkpointing and restart (v0.4.0)**

6.4.4 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 ex**
  - **Slurm**
  - **Condor**
  - **Cobalt**
  - **PBS/Torque**
  - **AWS**
  - **Azure**
  - **Nova/OpenStack/Jetstream (partial support)**
- **Support for launcher mechanisms**
  - **srun**
  - **aprun (Partial support on Crays and Theta as of 0.4.0)**
  - Various MPI launch mechanisms (Mpiexec, mpirun..)
- Support for remote execution using **SSH** and OAuth-based authentication (SSH execution support added in 0.3.0)
- **Utilizing multiple sites for a single script's execution (v0.4.0)**

- 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
- 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 (v0.5.0)**

6.4.5 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**

6.4.6 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)

6.4.7 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
- **Anonymous Usage Tracking (v0.4.0)**

6.4.8 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

6.5 Developer Guide

Parsl is a Parallel Scripting Library, designed to enable efficient workflow execution.

6.5.1 Importing

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

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

6.5.2 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

6.5.3 Apps

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

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 interpreter 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=None*, *walltime=60*, *sites='all'*, *cache=False*, *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 interpreter sees the definition of the decorated function, and the __call__ () which is invoked when a decorated function is called by the user.

PythonApp

Concrete subclass of AppBase that implements the Python App functionality.

class parsl.app.python_app.**PythonApp**(*func*, *executor=None*, *walltime=60*, *cache=False*, *sites='all'*, *fn_hash=None*)

Extends AppBase to cover the Python App.

BashApp

Concrete subclass of AppBase that implements the Bash App functionality.

class parsl.app.bash_app.**BashApp**(*func*, *executor=None*, *walltime=60*, *cache=False*, *sites='all'*, *fn_hash=None*)

6.5.4 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.

AppFutures

class parsl.dataflow.futures.**AppFuture**(*parent*, *tid=None*, *stdout=None*, *stderr=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.

__init__(*parent*, *tid=None*, *stdout=None*, *stderr=None*)

Initialize the AppFuture.

Args:

- parent (Future) : The parent future if one exists A default value of None should be passed in if app is not launched

KWargs:

- tid (Int) : Task id should be any unique identifier. Now Int.
- **stdout (str)** [Stdout file of the app.] Default: None
- **stderr (str)** [Stderr file of the app.] Default: None

__repr__()
Return repr(self).

add_done_callback(*fn*)
Attaches a callable that will be called when the future finishes.

Args:

fn: A callable that will be called with this future as its only argument when the future completes or is cancelled. The callable will always be called by a thread in the same process in which it was added. If the future has already completed or been cancelled then the callable will be called immediately. These callables are called in the order that they were added.

cancel()
Cancel the future if possible.

Returns True if the future was cancelled, False otherwise. A future cannot be cancelled if it is running or has already completed.

cancelled()
Return True if the future was cancelled.

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

exception(*timeout=None*)
Return the exception raised by the call that the future represents.

Args:

timeout: The number of seconds to wait for the exception if the future isn't done. If None, then there is no limit on the wait time.

Returns: The exception raised by the call that the future represents or None if the call completed without raising.

Raises: CancelledError: If the future was cancelled. TimeoutError: If the future didn't finish executing before the given

timeout.

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()

result(*timeout=None*)
Result.

Waits for the result of the AppFuture KWargs:

timeout (int): Timeout in seconds

running()
Return True if the future is currently executing.

update_parent(*fut*)
Add a callback to the parent to update the state.

This handles the case where the user has called result on the AppFuture before the parent exists.

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.

__init__(*fut*, *file_obj*, *parent=None*, *tid=None*)
Construct the DataFuture object.

If the file_obj is a string convert to a File.

Args:

- fut (AppFuture) : AppFuture that this DataFuture will track
- file_obj (string/File obj) : Something representing file(s)

Kwargs:

- parent ()
- tid (task_id) : Task id that this DataFuture tracks

__repr__()
Return repr(self).

add_done_callback(*fn*)
Attaches a callable that will be called when the future finishes.

Args:

fn: A callable that will be called with this future as its only argument when the future completes or is cancelled. The callable will always be called by a thread in the same process in which it was added. If the future has already completed or been cancelled then the callable will be called immediately. These callables are called in the order that they were added.

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

Note: This may not work

cancelled()
Return True if the future was cancelled.

done()
Return True of the future was cancelled or finished executing.

exception(*timeout=None*)
Return the exception raised by the call that the future represents.

Args:

timeout: The number of seconds to wait for the exception if the future isn't done. If None, then there is no limit on the wait time.

Returns: The exception raised by the call that the future represents or None if the call completed without raising.

Raises: CancelledError: If the future was cancelled. TimeoutError: If the future didn't finish executing before the given

timeout.

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:

- parent_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.

running()
Return True if the future is currently executing.

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

6.5.5 Exceptions

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

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

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

This is basically a type error.

class parsl.app.errors.**InvalidAppTypeError**
An invalid app type was requested from 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.**DataFlowException**
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.

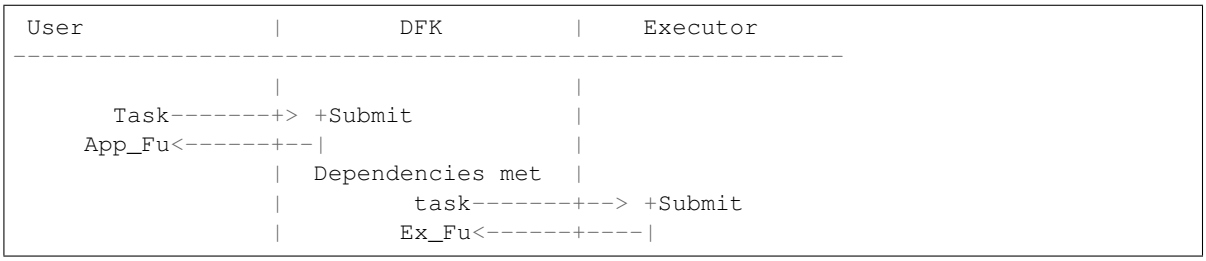
Deprecated.

6.5.6 DataFlowKernel

class parsl.dataflow.dflow.**DataFlowKernel**(*config=None*, *executors=None*, *lazyErrors=True*, *appCache=True*, *rundir=None*, *retries=0*, *checkpointFiles=None*, *checkpointMode=None*)
The DataFlowKernel adds dependency awareness to an existing executor.

It is responsible for managing futures, such that when dependencies are resolved, pending tasks move to the runnable state.

Here's a simplified diagram of what happens internally:



__init__(*config=None*, *executors=None*, *lazyErrors=True*, *appCache=True*, *rundir=None*, *retries=0*, *checkpointFiles=None*, *checkpointMode=None*)
Initialize the DataFlowKernel.

Please note that keyword args passed to the DFK here will always override options passed in via the config.

KWargs:

- config (dict): A single data object encapsulating all config attributes
- executors (list of Executor objs): Optional, kept for (somewhat) backward compatibility with 0.2.0
- lazyErrors(bool): Default=True, allow workflow to continue on app failures.
- appCache (bool): Enable caching of apps
- rundir (str): Path to run directory. Defaults to ./runinfo/runNNN
- retries(int): Default=0, Set the number of retry attempts in case of failure
- checkpointFiles (list of str): List of filepaths to checkpoint files
- checkpointMode (None, 'dfk_exit', 'task_exit', 'periodic'): Method to use.

Returns: DataFlowKernel object

__weakref__
list of weak references to the object (if defined)

checkpoint(*tasks=None*)
Checkpoint the dfk incrementally to a checkpoint file.

When called, every task that has been completed yet not checkpointed is checkpointed to a file.

Kwargs:

- **tasks (List of task ids)** [List of task ids to checkpoint. Default=None] if set to None, we iterate over all tasks held by the DFK.

Note: Checkpointing only works if memoization is enabled

Returns: Checkpoint dir if checkpoints were written successfully. By default the checkpoints are written to the RUNDIR of the current run under RUNDIR/checkpoints/{tasks.pkl, dfk.pkl}

cleanup()
DataFlowKernel cleanup.

This involves killing resources explicitly and sending die messages to IPP workers.

If the executors are managed, i.e created by the DFK then : we scale_in each of the executors and call executor.shutdown else : we do nothing. Executor cleanup is left to the user.

config
Returns the fully initialized config that the DFK is actively using.

DO *NOT* update.

Returns:

- config (dict)

handle_update(*task_id*, *future*, *memo_cbk=False*)
This function is called only as a callback from a task being done.

Move done task from runnable -> done Move newly doable tasks from pending -> runnable , and launch

Args: task_id (string) : Task id which is a uuid string future (Future) : The future object corresponding to the task which makes this callback

KWargs: memo_cbk(Bool) : Indicates that the call is coming from a memo update, that does not require additional memo updates.

launch_task(*task_id*, *executable*, **args*, ***kwargs*)
Handle the actual submission of the task to the executor layer.

Args: task_id (uuid string) : A uuid string that uniquely identifies the task executable (callable) : A callable object args (list of positional args) kwargs (arbitrary keyword arguments)

Returns: Future that tracks the execution of the submitted executable

load_checkpoints(*checkpointDirs*)
Load checkpoints from the checkpoint files into a dictionary.

The results are used to pre-populate the memoizer's lookup_table

Kwargs:

- checkpointDirs (list) : List of run folder to use as checkpoints Eg. [‘runinfo/001’, ‘runinfo/002’]

Returns:

- dict containing, hashed -> future mappings

static sanitize_and_wrap(*task_id*, *args*, *kwargs*)
This function should be called **ONLY** when all the futures we track have been resolved.

If the user hid futures a level below, we will not catch it, and will (most likely) result in a type error .

Args: task_id (uuid str) : Task id func (Function) : App function args (List) : Positional args to app function kwargs (Dict) : Kwargs to app function

Return: partial Function evaluated with all dependencies in args, kwargs and kwargs[‘inputs’] evaluated.

submit(*func*, **args*, *parsl_sites='all'*, *fn_hash=None*, *cache=False*, ***kwargs*)
Add task to the dataflow system.

If the app task has the sites attributes not set (default=='all') the task will be launched on a randomly selected executor from the list of executors. This behavior could later be updated to support binding to sites based on user specified criteria.

If the app task specifies a particular set of sites, it will be targetted at those specific sites.

```
>>> IF all deps are met:
>>>   send to the runnable queue and launch the task
>>> ELSE:
>>>   post the task in the pending queue
```

Args:

- func : A function object
- *args : Args to the function

KWargs :

- **parsl_sites (List|String)** [List of sites this call could go to.] Default='all'

- **fn_hash (Str)** [Hash of the function and inputs] Default=None
- cache (Bool) : To enable memoization or not
- kwargs (dict) : Rest of the kwargs to the fn passed as dict.

Returns: (AppFuture) [DataFutures,]

6.5.7 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.

ParslExecutor (Abstract Base Class)

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 example).

__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 coroutine, 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

Specify if scaling is enabled.

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

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

Submit.

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

ThreadPoolExecutor

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

The thread pool executor.

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

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

scaling_enabled

Specify if scaling is enabled.

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

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

IPyParallelExecutor

class parsl.executors.ipp.**IPyParallelExecutor**(*execution_provider=None*, *reuse_controller=True*, *engine_json_file='~/.ipython/profile_default/security/ipcontroller-engine.json'*, *engine_dir='.'*, *controller=None*, *config=None*)

The IPython Parallel executor.

This executor allows us to take advantage of multiple processes running locally or remotely via IPythonParallel's pilot execution system.

Note: Some deficiencies with this executor are:

1. Ipengine's execute one task at a time. This means one engine per core is necessary to exploit the full parallelism of a node.
2. No notion of remaining walltime.
3. Lack of throttling means tasks could be queued up on a worker.

__init__ (*execution_provider=None*, *reuse_controller=True*, *engine_json_file='~/.ipython/profile_default/security/ipcontrolle engine.json'*, *engine_dir='.'*, *controller=None*, *config=None*)

Initialize the IPyParallel pool. The initialization takes all relevant parameters via KWargs.

Note: If initBlocks > 0, and a scalable execution_provider is attached, then the provider will be initialized here.

Args:

- self

KWargs:

- execution_provider (ExecutionProvider object)
- reuse_controller (Bool) : If True ipp executor will attempt to connect to an available controller. Default: True
- engine_json_file (str): Path to json engine file that will be used to compose ipp launch commands at scaling events. Default : '~/.ipython/profile_default/security/ipcontroller-engine.json'
- engine_dir (str) : Alternative to above, specify the engine_dir
- config (dict). Default: '.'

compose_launch_cmd (*filepath*, *engine_dir*, *container_image*)

Reads the json contents from filepath and uses that to compose the engine launch command.

Args: filepath: Path to the engine file engine_dir : CWD for the engines .

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

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

Scales out the number of active workers by 1.

This method is notImplemented for threads and will raise the error if called.

scaling_enabled

Specify if scaling is enabled.

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

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

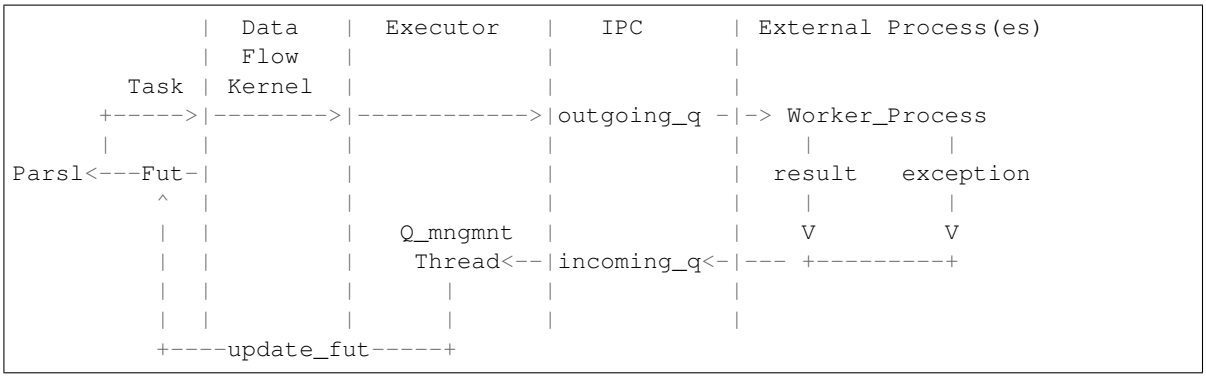
Swift/Turbine Executor

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

The Turbine executor.

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

Here is a diagram



__init__(*swift_attribs=None*, *config=None*, ***kwargs*)
Initialize the thread pool.

Trying to implement the emews model.

Kwargs:

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

_queue_management_worker()
Listen to the queue for task status messages and handle them.

Depending on the message, tasks will be updated with results, exceptions, or 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 do not support these yet, but they could be added easily.

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

The *None* message is a die request.

_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: NotImplementedError

scale_out(*workers=1*)
Scales out the number of active workers by 1.

This method is not implemented for threads and will raise the error if called. This would be nice to have, and can be done

Raises: NotImplementedError

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

submit(*func*, **args*, ***kwargs*)
Submits work to the the outgoing_q.

The outgoing_q is 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

6.5.8 Execution Providers

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

ExecutionProvider (Base)

class libsubmit.providers.provider_base.**ExecutionProvider**

Define the strict interface for all Execution Provider

```
                      +------------------
                      |
script_string ------->|  submit
     id      <--------|---+
                      |
[ ids ]       ------->|  status
[statuses]   <--------|----+
                      |
[ ids ]       ------->|  cancel
[cancel]     <--------|----+
                      |
[True/False] <--------|  scaling_enabled
                      |
                      +-------------------
```

__weakref__

list of weak references to the object (if defined)

cancel(*job_ids*)

Cancels the resources identified by the job_ids provided by the user.

Args:

- job_ids (list): A list of job identifiers

Returns:

- A list of status from cancelling the job which can be True, False

Raises:

- ExecutionProviderExceptions or its subclasses

channels_required
Does the execution provider require a channel to function. Generally all Cloud api's require no channels while all bash script based systems such as schedulers for campus clusters (slurm, torque, cobalt, condor..) need channels

Returns:

- Status (Bool)

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

Returns:

- Status (Bool)

status(*job_ids*)
Get the status of a list of jobs identified by the job identifiers returned from the submit request.

Args:

- job_ids (list) : A list of job identifiers

Returns:

- A list of status from ['PENDING', 'RUNNING', 'CANCELLED', 'COMPLETED', 'FAILED', 'TIMEOUT'] corresponding to each job_id in the job_ids list.

Raises:

- ExecutionProviderExceptions or its subclasses

submit(*cmd_string*, *blocksize*, *job_name='parsl.auto'*)
The submit method takes the command string to be executed upon instantiation of a resource most often to start a pilot (such as IPP engine or even Swift-T engines).

Args :

- cmd_string (str) : The bash command string to be executed.
- blocksize (int) : Blocksize to be requested

KWargs:

- job_name (str) : Human friendly name to be assigned to the job request

Returns:

- A job identifier, this could be an integer, string etc

Raises:

- ExecutionProviderExceptions or its subclasses

Local

class libsubmit.providers.local.local.**Local**(*config*, *channel_script_dir=None*, *channel=None*)
Local Execution Provider

This provider is used to launch IPP engines on the localhost.

Warning: Please note that in the config documented below, description and values are placed inside a schema that is delimited by #{ schema.. }

Here's the scheme for the Local provider:

```
{ "execution" : { # Definition of all execution aspects of a site

     "executor"   : #{Description: Define the executor used as task executor,
                    # Type : String,
                    # Expected : "ipp",
                    # Required : True},

     "provider"   : #{Description : The provider name, in this case local
                    # Type : String,
                    # Expected : "local",
                    # Required :  True },

     "scriptDir"  : #{Description : Relative or absolute path to a
                    # directory in which intermediate scripts are placed
                    # Type : String,
                    # Default : "./.scripts"},

     "block" : { # Definition of a block

         "initBlocks" : #{Description : # of blocks to provision at the start of
                        # the DFK
                        # Type : Integer
                        # Default : ?
                        # Required :    },

         "minBlocks" :  #{Description : Minimum # of blocks outstanding at any␣
↪time
                        # WARNING :: Not Implemented
                        # Type : Integer
                        # Default : 0 },

         "maxBlocks" :  #{Description : Maximum # Of blocks outstanding at any␣
↪time
                        # WARNING :: Not Implemented
                        # Type : Integer
                        # Default : ? },
     }
   }
}
```

__init__(*config*, *channel_script_dir=None*, *channel=None*)
Initialize the local provider class

Args:

- Config (dict): Dictionary with all the config options.

__repr__()
Return repr(self).

cancel(*job_ids*)

Cancels the jobs specified by a list of job ids

Args: job_ids : [<job_id> . . .]

Returns : [True/False. . .] : If the cancel operation fails the entire list will be False.

channels_required
Does the execution provider require a channel to function. Generally all Cloud api's require no channels while all bash script based systems such as schedulers for campus clusters (slurm, torque, cobalt, condor..) need channels

Returns:

- Status (Bool)

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

Returns:

- Status (Bool)

status(*job_ids*)
Get the status of a list of jobs identified by their ids.

Args:

- job_ids (List of ids) : List of identifiers for the jobs

Returns:

- List of status codes.

submit(*cmd_string*, *blocksize*, *job_name='parsl.auto'*)
Submits the cmd_string onto an Local Resource Manager job of blocksize parallel elements. Submit returns an ID that corresponds to the task that was just submitted.

If tasks_per_node < 1: 1/tasks_per_node is provisioned

If tasks_per_node == 1: A single node is provisioned

If tasks_per_node > 1 : tasks_per_node * blocksize number of nodes are provisioned.

Args:

- cmd_string :(String) Commandline invocation to be made on the remote side.
- blocksize :(float) - Not really used for local

Kwargs:

- job_name (String): Name for job, must be unique

Returns:

- None: At capacity, cannot provision more
- job_id: (string) Identifier for the job

Slurm

class libsubmit.providers.slurm.slurm.**Slurm**(*config*, *channel=None*)
Slurm Execution Provider

This provider uses sbatch to submit, squeue for status and scancel to cancel jobs. The sbatch script to be used is created from a template file in this same module.

Warning: Please note that in the config documented below, description and values are placed inside a schema that is delimited by <{ schema.. }>

Here's a sample config for the Slurm provider:

```
{ "execution" : { # Definition of all execution aspects of a site

    "executor"   : #{Description: Define the executor used as task executor,
                   # Type : String,
                   # Expected : "ipp",
                   # Required : True},

    "provider"   : #{Description : The provider name, in this case slurm
                   # Type : String,
                   # Expected : "slurm",
                   # Required :  True },

    "scriptDir"  : #{Description : Relative or absolute path to a
                   # directory in which intermediate scripts are placed
                   # Type : String,
                   # Default : "./.scripts"},

    "block" : { # Definition of a block

        "nodes"      : #{Description : # of nodes to provision per block
                       # Type : Integer,
                       # Default: 1},

        "taskBlocks" : #{Description : # of workers to launch per block
                       # as either an number or as a bash expression.
                       # for eg, "1" , "$(($CORES / 2))"
                       # Type : String,
                       #  Default: "1" },

        "walltime"  :  #{Description : Walltime requested per block in HH:MM:SS
                       # Type : String,
                       # Default : "00:20:00" },

        "initBlocks" : #{Description : # of blocks to provision at the start of
                       # the DFK
                       # Type : Integer
                       # Default : ?
                       # Required :    },

        "minBlocks" :  #{Description : Minimum # of blocks outstanding at any␣
↪time
                       # WARNING :: Not Implemented
                       # Type : Integer
                       # Default : 0 },

        "maxBlocks" :  #{Description : Maximum # Of blocks outstanding at any␣
↪time
                       # WARNING :: Not Implemented
```

(continues on next page)

(continued from previous page)

```
                        # Type : Integer
                        # Default : ? },

        "options"   : {  # Scheduler specific options

            "partition" : #{Description : Slurm partition to request blocks from
                          # Type : String,
                          # Required : True },

            "overrides" : #{"Description : String to append to the #SBATCH blocks
                          # in the submit script to the scheduler
                          # Type : String,
                          # Required : False },
        }
    }
  }
}
```

__init__(*config*, *channel=None*)
Initialize the Slurm class

Args:

- Config (dict): Dictionary with all the config options.

KWargs:

- Channel (None): A channel is required for slurm.

__repr__()
Return repr(self).

cancel(*job_ids*)
Cancels the jobs specified by a list of job ids

Args: job_ids : [<job_id> . . .]

Returns : [True/False. . .] : If the cancel operation fails the entire list will be False.

channels_required
Returns Bool on whether a channel is required

current_capacity
Returns the current blocksize. This may need to return more information in the futures : { minsize, maxsize, current_requested }

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

Returns:

- Status (Bool)

status(*job_ids*)
Get the status of a list of jobs identified by their ids.

Args:

- job_ids (List of ids) : List of identifiers for the jobs

Returns:

- List of status codes.

submit (*cmd_string*, *blocksize*, *job_name='parsl.auto'*)

Submits the cmd_string onto an Local Resource Manager job of blocksize parallel elements. Submit returns an ID that corresponds to the task that was just submitted.

If tasks_per_node < 1 : ! This is illegal. tasks_per_node should be integer

If tasks_per_node == 1: A single node is provisioned

If tasks_per_node > 1 : tasks_per_node * blocksize number of nodes are provisioned.

Args:

- cmd_string :(String) Commandline invocation to be made on the remote side.
- blocksize :(float)

Kwargs:

- job_name (String): Name for job, must be unique

Returns:

- None: At capacity, cannot provision more
- job_id: (string) Identifier for the job

Cobalt

class libsubmit.providers.cobalt.cobalt.**Cobalt** (*config*, *channel=None*)

Cobalt Execution Provider

This provider uses cobalt to submit (qsub), obtain the status of (qstat), and cancel (qdel) jobs. Theo script to be used is created from a template file in this same module.

Warning: Please note that in the config documented below, description and values are placed inside a schema that is delimited by #{ schema.. }

Here's the scheme for the Cobalt provider:

```
{ "execution" : { # Definition of all execution aspects of a site

     "executor"   : #{Description: Define the executor used as task executor,
                    # Type : String,
                    # Expected : "ipp",
                    # Required : True},

     "provider"   : #{Description : The provider name, in this case cobalt
                    # Type : String,
                    # Expected : "cobalt",
                    # Required :  True },

     "launcher"   : #{Description : Launcher to use for launching workers
                    # it is often necessary to use a launcher that the scheduler␣
↪supports to
                    # launch workers on multi-node jobs, or to partition MPI jobs
                    # Type : String,
                    # Default : "singleNode" },
```

(continues on next page)

(continued from previous page)

```
    "scriptDir"  : #{Description : Relative or absolute path to a
                   # directory in which intermediate scripts are placed
                   # Type : String,
                   # Default : "./.scripts"},

    "block" : { # Definition of a block

        "nodes"      : #{Description : # of nodes to provision per block
                       # Type : Integer,
                       # Default: 1},

        "taskBlocks" : #{Description : # of workers to launch per block
                       # as either an number or as a bash expression.
                       # for eg, "1" , "$(($CORES / 2))"
                       # Type : String,
                       #  Default: "1" },

        "walltime"  :  #{Description : Walltime requested per block in HH:MM:SS
                       # Type : String,
                       # Default : "01:00:00" },

        "initBlocks" : #{Description : # of blocks to provision at the start of
                       # the DFK
                       # Type : Integer
                       # Default : ?
                       # Required :    },

        "minBlocks" :  #{Description : Minimum # of blocks outstanding at any␣
↪time
                       # WARNING :: Not Implemented
                       # Type : Integer
                       # Default : 0 },

        "maxBlocks" :  #{Description : Maximum # Of blocks outstanding at any␣
↪time
                       # WARNING :: Not Implemented
                       # Type : Integer
                       # Default : ? },

        "options"   : {  # Scheduler specific options

            "account"   : #{Description : Account that the job will be charged␣
↪against
                          # Type : String,
                          # Required : True },

            "queue"     : #{Description : Torque queue to request blocks from
                          # Type : String,
                          # Required : False },

            "overrides" : #{"Description : String to append to the Torque submit␣
↪script
                          # in the submit script to the scheduler
                          # Type : String,
                          # Required : False },
        }
```

(continues on next page)

(continued from previous page)

```
        }
    }
}
```

__init__ (*config*, *channel=None*)
Initialize the Cobalt execution provider class

Args:

- Config (dict): Dictionary with all the config options.

KWargs :

- channel (channel object) : default=None A channel object

__repr__ ()
Return repr(self).

cancel (*job_ids*)
Cancels the jobs specified by a list of job ids

Args: job_ids : [<job_id> . . .]

Returns : [True/False. . .] : If the cancel operation fails the entire list will be False.

channels_required
Returns Bool on whether a channel is required

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

Returns:

- Status (Bool)

status (*job_ids*)
Get the status of a list of jobs identified by their ids.

Args:

- job_ids (List of ids) : List of identifiers for the jobs

Returns:

- List of status codes.

submit (*cmd_string*, *blocksize*, *job_name='parsl.auto'*)
Submits the cmd_string onto an Local Resource Manager job of blocksize parallel elements. Submit returns an ID that corresponds to the task that was just submitted.

If tasks_per_node < 1 : ! This is illegal. tasks_per_node should be integer

If tasks_per_node == 1: A single node is provisioned

If tasks_per_node > 1 : tasks_per_node * blocksize number of nodes are provisioned.

Args:

- cmd_string :(String) Commandline invocation to be made on the remote side.
- blocksize :(float)

Kwargs:

- job_name (String): Name for job, must be unique

Returns:

- None: At capacity, cannot provision more
- job_id: (string) Identifier for the job

Condor

class libsubmit.providers.condor.condor.**Condor**(*config*, *channel=None*)
Condor Execution Provider

> **Warning:** Please note that in the config documented below, description and values are placed inside a schema that is delimited by #{ schema.. }

Here's the schema for the Condor provider:

```
{ "execution" : { # Definition of all execution aspects of a site

     "executor"   : #{Description: Define the executor used as task executor,
                    # Type : String,
                    # Expected : "ipp",
                    # Required : True},

     "provider"   : #{Description : The provider name, in this case condor
                    # Type : String,
                    # Expected : "condor",
                    # Required :  True },

     "launcher"   : #{Description : Launcher to use for launching workers
                    # Since condor doesn't generally do multi-node, "singleNode"␣
↪is the
                    # only meaningful launcher.
                    # Type : String,
                    # Default : "singleNode" },

     "scriptDir"  : #{Description : Relative or absolute path to a
                    # directory in which intermediate scripts are placed
                    # Type : String,
                    # Default : "./.scripts"},

     "block" : { # Definition of a block

         "nodes"      : #{Description : # of nodes to provision per block
                        # Type : Integer,
                        # Default: 1},

         "taskBlocks" : #{Description : # of workers to launch per block
                        # as either an number or as a bash expression.
                        # for eg, "1" , "$(($CORES / 2))"
                        # Type : String,
                        #  Default: "1" },

         "walltime"  :  #{Description : Walltime requested per block in HH:MM:SS
                        # Type : String,
                        # Default : "01:00:00" },
```

(continues on next page)

(continued from previous page)

```
        "initBlocks" : #{Description : # of blocks to provision at the start of
                       # the DFK
                       # Type : Integer
                       # Default : ?
                       # Required :    },

        "minBlocks" :  #{Description : Minimum # of blocks outstanding at any␣
↪time
                       # WARNING :: Not Implemented
                       # Type : Integer
                       # Default : 0 },

        "maxBlocks" :  #{Description : Maximum # Of blocks outstanding at any␣
↪time
                       # WARNING :: Not Implemented
                       # Type : Integer
                       # Default : ? },

        "options"   : {  # Scheduler specific options

            "project"    : #{Description : Project to which the job will be␣
↪charged against
                           # Type : String,
                           # Required : True },

            "overrides"  : #{"Description : String to add specific condor␣
↪attributes to the
                           # Condor submit script
                           # Type : String,
                           # Required : False },

            "workerSetup": #{"Description : String that sets up the env for the␣
↪workers as well
                           # apps to run
                           # Type : String,
                           # Required : False },

            "requirements": #{"Description : Condor requirements
                            # Type : String,
                            # Required : True },
        }
    }
  }
}
```

__init__(*config*, *channel=None*)
Initialize the Condor class

Args:

- Config (dict): Dictionary with all the config options.

KWargs:

- Channel (none): A channel is required for htcondor.

__repr__()
Return repr(self).

cancel (*job_ids*)
Cancels the jobs specified by a list of job ids

Args: job_ids : [<job_id> . . .]

Returns : [True/False. . .] : If the cancel operation fails the entire list will be False.

channels_required
Returns Bool on whether a channel is required

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

Returns:

- Status (Bool)

status (*job_ids*)
Get the status of a list of jobs identified by their ids.

Args:

- job_ids (List of ids) : List of identifiers for the jobs

Returns:

- List of status codes.

submit (*cmd_string*, *blocksize*, *job_name='parsl.auto'*)
Submits the cmd_string onto an Local Resource Manager job of blocksize parallel elements.

example file with the complex case of multiple submits per job: Universe =vanilla output = out.$(Cluster).$(Process) error = err.$(Cluster).$(Process) log = log.$(Cluster) leave_in_queue = true executable = test.sh queue 5 executable = foo queue 1

$ condor_submit test.sub Submitting job(s)..... 5 job(s) submitted to cluster 118907. 1 job(s) submitted to cluster 118908.

Torque

class libsubmit.providers.torque.torque.**Torque**(*config*, *channel=None*)
Torque Execution Provider

This provider uses sbatch to submit, squeue for status, and scancel to cancel jobs. The sbatch script to be used is created from a template file in this same module.

Warning: Please note that in the config documented below, description and values are placed inside a schema that is delimited by #{ schema.. }

Here's the scheme for the Torque provider:

```
{ "execution" : { # Definition of all execution aspects of a site

     "executor"   : #{Description: Define the executor used as task executor,
                    # Type : String,
                    # Expected : "ipp",
                    # Required : True},
```

(continues on next page)

(continued from previous page)

```
    "provider"    : #{Description : The provider name, in this case torque
                    # Type : String,
                    # Expected : "torque",
                    # Required :  True },

    "scriptDir"   : #{Description : Relative or absolute path to a
                    # directory in which intermediate scripts are placed
                    # Type : String,
                    # Default : "./scripts"},

    "block" : { # Definition of a block

        "nodes"       : #{Description : # of nodes to provision per block
                        # Type : Integer,
                        # Default: 1},

        "taskBlocks" : #{Description : # of workers to launch per block
                        # as either an number or as a bash expression.
                        # for eg, "1" , "$(($CORES / 2))"
                        # Type : String,
                        #  Default: "1" },

        "walltime"  :  #{Description : Walltime requested per block in HH:MM:SS
                        # Type : String,
                        # Default : "00:20:00" },

        "initBlocks" : #{Description : # of blocks to provision at the start of
                        # the DFK
                        # Type : Integer
                        # Default : ?
                        # Required :    },

        "minBlocks" :  #{Description : Minimum # of blocks outstanding at any␣
↪time
                        # WARNING :: Not Implemented
                        # Type : Integer
                        # Default : 0 },

        "maxBlocks" :  #{Description : Maximum # Of blocks outstanding at any␣
↪time
                        # WARNING :: Not Implemented
                        # Type : Integer
                        # Default : ? },

        "options"   : {  # Scheduler specific options

            "account"   : #{Description : Account the job will be charged against
                          # Type : String,
                          # Required : True },

            "queue"     : #{Description : Torque queue to request blocks from
                          # Type : String,
                          # Required : False },

            "overrides" : #{"Description : String to append to the Torque submit␣
↪script
                          # in the submit script to the scheduler
```

(continues on next page)

(continued from previous page)

```
                                # Type : String,
                                # Required : False },
            }
        }
    }
}
```

__init__ (*config*, *channel=None*)
Initialize the Torque class

Args:

- Config (dict): Dictionary with all the config options.

KWargs:

- Channel (None): A channel is required for torque.

__repr__ ()
Return repr(self).

cancel (*job_ids*)
Cancels the jobs specified by a list of job ids

Args: job_ids : [<job_id> . . .]

Returns : [True/False. . .] : If the cancel operation fails the entire list will be False.

channels_required
Returns Bool on whether a channel is required

current_capacity
Returns the current blocksize. This may need to return more information in the futures : { minsize, maxsize, current_requested }

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

Returns:

- Status (Bool)

status (*job_ids*)
Get the status of a list of jobs identified by their ids.

Args:

- job_ids (List of ids) : List of identifiers for the jobs

Returns:

- List of status codes.

submit (*cmd_string*, *blocksize*, *job_name='parsl.auto'*)
Submits the cmd_string onto an Local Resource Manager job of blocksize parallel elements. Submit returns an ID that corresponds to the task that was just submitted.

If tasks_per_node < 1 : ! This is illegal. tasks_per_node should be integer

If tasks_per_node == 1: A single node is provisioned

If tasks_per_node > 1 : tasks_per_node * blocksize number of nodes are provisioned.

Args:

- cmd_string :(String) Commandline invocation to be made on the remote side.
- blocksize :(float)

Kwargs:

- job_name (String): Name for job, must be unique

Returns:

- None: At capacity, cannot provision more
- job_id: (string) Identifier for the job

GridEngine

Amazon Web Services

class libsubmit.providers.aws.aws.**EC2Provider**(*config*, *channel=None*)

Here's a sample config for the EC2 provider:

```
{ "auth" : { # Definition of authentication method for AWS. One of 3 methods are␣
↪required to authenticate
             # with AWS : keyfile, profile or env_variables. If keyfile or␣
↪profile is not set Boto3 will
             # look for the following env variables :
             # AWS_ACCESS_KEY_ID : The access key for your AWS account.
             # AWS_SECRET_ACCESS_KEY : The secret key for your AWS account.
             # AWS_SESSION_TOKEN : The session key for your AWS account.

    "keyfile"    : #{Description: Path to json file that contains 'AWSAccessKeyId
↪' and 'AWSSecretKey'
                   # Type : String,
                   # Required : False},

    "profile"    : #{Description: Specify the profile to be used from the␣
↪standard aws config file
                   # ~/.aws/config.
                   # Type : String,
                   # Expected : "default", # Use the 'default' aws profile
                   # Required : False},

  },

  "execution" : { # Definition of all execution aspects of a site

    "executor"   : #{Description: Define the executor used as task executor,
                   # Type : String,
                   # Expected : "ipp",
                   # Required : True},

    "provider"   : #{Description : The provider name, in this case ec2
                   # Type : String,
                   # Expected : "aws",
                   # Required :  True },

    "block" : { # Definition of a block

        "nodes"      : #{Description : # of nodes to provision per block
```

(continues on next page)

(continued from previous page)

```
                        # Type : Integer,
                        # Default: 1},

        "taskBlocks" : #{Description : # of workers to launch per block
                        # as either an number or as a bash expression.
                        # for eg, "1" , "$(($CORES / 2))"
                        # Type : String,
                        #  Default: "1" },

        "walltime"   :  #{Description : Walltime requested per block in HH:MM:SS
                        # Type : String,
                        # Default : "00:20:00" },

        "initBlocks" : #{Description : # of blocks to provision at the start of
                        # the DFK
                        # Type : Integer
                        # Default : ?
                        # Required :    },

        "minBlocks" :  #{Description : Minimum # of blocks outstanding at any␣
↪time
                        # WARNING :: Not Implemented
                        # Type : Integer
                        # Default : 0 },

        "maxBlocks" :  #{Description : Maximum # Of blocks outstanding at any␣
↪time
                        # WARNING :: Not Implemented
                        # Type : Integer
                        # Default : ? },

        "options"   : {  # Scheduler specific options


            "instanceType" : #{Description : Instance type t2.small|t2...
                             # Type : String,
                             # Required : False
                             # Default : t2.small },

            "imageId"      : #{"Description : String to append to the #SBATCH␣
↪blocks
                             # in the submit script to the scheduler
                             # Type : String,
                             # Required : False },

            "region"       : #{"Description : AWS region to launch machines in
                             # in the submit script to the scheduler
                             # Type : String,
                             # Default : 'us-east-2',
                             # Required : False },

            "keyName"      : #{"Description : Name of the AWS private key (.pem␣
↪file)
                             # that is usually generated on the console to allow␣
↪ssh access
                             # to the EC2 instances, mostly for debugging.
                             # in the submit script to the scheduler
```

(continues on next page)

(continued from previous page)

```
                                # Type : String,
                                # Required : True },

            "spotMaxBid"    : #{"Description : If requesting spot market machines,
↪ specify
                                # the max Bid price.
                                # Type : Float,
                                # Required : False },
        }
    }
  }
}
```

__init__ (*config*, *channel=None*)
Initialize the EC2Provider class

Args:

- Config (dict): Dictionary with all the config options.

KWargs:

- Channel (None): A channel is not required for EC2.

__repr__ ()
Return repr(self).

cancel (*job_ids*)
Cancels the jobs specified by a list of job ids

Args: job_ids (list) : List of of job identifiers

Returns : [True/False...] : If the cancel operation fails the entire list will be False.

channels_required
Does the execution provider require a channel to function. Generally all Cloud api's require no channels while all bash script based systems such as schedulers for campus clusters (slurm, torque, cobalt, condor..) need channels

Returns:

- Status (Bool)

config_route_table (*vpc*, *internet_gateway*)
Configure route table for vpc

[TODO] Args:

- vpc (type) : ?
- vpc (type) : ?

create_session ()
Here we will first look in the ~/.aws/config file.

First we look in config["auth"]["keyfile"] for a path to a json file with the credentials. the keyfile should have 'AWSAccessKeyId' and 'AWSSecretKey'

Next we look for config["auth"]["profile"] for a profile name and try to use the Session call to auto pick up the keys for the profile from the user default keys file ~/.aws/config.

Lastly boto3 will look for the keys in env variables: AWS_ACCESS_KEY_ID : The access key for your AWS account. AWS_SECRET_ACCESS_KEY : The secret key for your AWS account.

AWS_SESSION_TOKEN : The session key for your AWS account. This is only needed when you are using temporary credentials. The AWS_SECURITY_TOKEN environment variable can also be used, but is only supported for backwards compatibility purposes. AWS_SESSION_TOKEN is supported by multiple AWS SDKs besides python.

create_vpc()
Create and configure VPC [TODO] Describe this a bit more ...

current_capacity
Returns the current blocksize. This may need to return more information in the futures : { minsize, maxsize, current_requested }

get_instance_state(*instances=None*)
Get stateus of all instances on EC2 which were started by this file

initialize_boto_client()
Use auth configs to initialize the boto client

pretty_configs(*configs*)
prettyprint config

read_configs(*config_file*)
Read config file

read_state_file(*statefile*)
If this script has been run previously, it will be persisitent by writing resource ids to state file. On run, the script looks for a state file before creating new infrastructure

scale_in(*size*)
Scale cluster in (smaller)

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

Returns:

- Status (Bool)

security_group(*vpc*)
Create and configure security group. Allows all ICMP in, all tcp and udp in within vpc

TODO : Open up only the necessary port ranges.

show_summary()
Print human readable summary of current AWS state to log and to console

shut_down_instance(*instances=None*)
Shuts down a list of instances if provided or the last instance started up if none provided

[TODO] ...

spin_up_instance(*cmd_string*, *job_name*)
Starts an instance in the vpc in first available subnet. Starts up n instances if nodes per block > 1 Not supported. We only do 1 node per block

Args:

- cmd_string (str) : Command string to execute on the node
- job_name (str) : Name associated with the instances

status(*job_ids*)
Get the status of a list of jobs identified by their ids.

Args:

- job_ids (List of ids) : List of identifiers for the jobs

Returns:

- List of status codes.

submit(*cmd_string='sleep 1'*, *blocksize=1*, *job_name='parsl.auto'*)
Submits the cmd_string onto a freshly instantiates AWS EC2 instance. Submit returns an ID that corresponds to the task that was just submitted.

Args:

- cmd_string (str): Commandline invocation to be made on the remote side.
- blocksize (int) : Number of blocks requested

Kwargs:

- job_name (String): Prefix for job name

Returns:

- None: At capacity, cannot provision more
- job_id: (string) Identifier for the job

teardown()
Terminate all EC2 instances, delete all subnets, delete security group, delete vpc and reset all instance variables

Azure

class libsubmit.providers.azure.azureProvider.**AzureProvider**(*config*)

__init__(*config*)
Initialize Azure provider. Uses Azure python SDK to provide execution resources

cancel()
Destroy an azure VM

status()
Get status of azure VM. Not implemented yet.

submit()
Uses AzureDeployer to spin up an instance and connect it to the iPyParallel controller

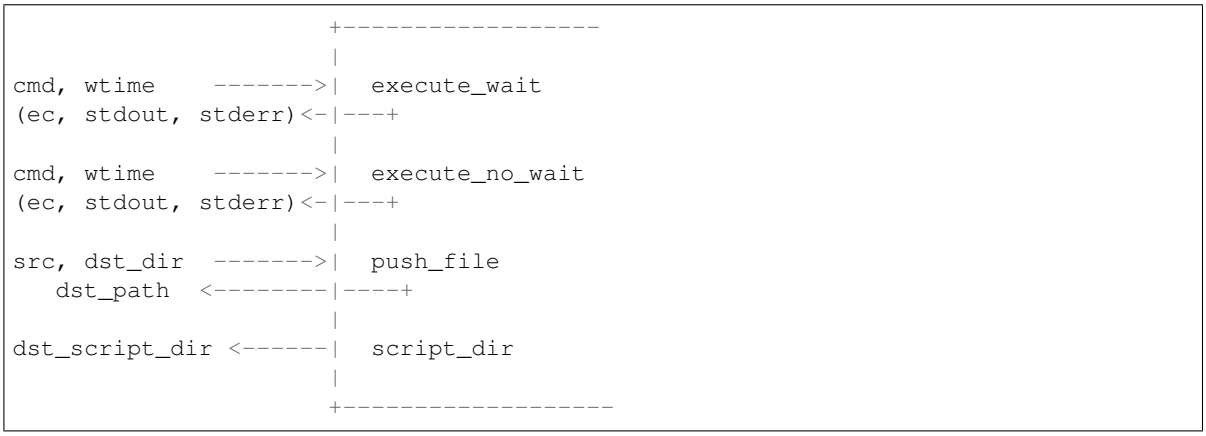
Google Cloud Platform

6.5.9 Channels

For certain resources such as campus clusters or supercomputers at research laboratories, resource requirements may require authentication. For instance, some resources may allow access to their job schedulers from only their login-nodes, which require you to authenticate on through SSH, GSI-SSH and sometimes even require two-factor authentication. Channels are simple abstractions that enable the ExecutionProvider component to talk to the resource managers of compute facilities. The simplest Channel, *LocalChannel*, simply executes commands locally on a shell, while the *SshChannel* authenticates you to remote systems.

class libsubmit.channels.channel_base.**Channel**

Define the interface to all channels. Channels are usually called via the execute_wait function. For channels that execute remotely, a push_file function allows you to copy over files.



__weakref__

list of weak references to the object (if defined)

close()

Closes the channel. Clean out any auth credentials.

Args: None

Returns: Bool

execute_no_wait(*cmd*, *walltime*, **args*, ***kwargs*)

Optional. THis is infrequently used.

Args:

- cmd (string): Command string to execute over the channel
- walltime (int) : Timeout in seconds

Returns:

- (exit_code(None), stdout, stderr) (int, io_thing, io_thing)

execute_wait(*cmd*, *walltime*, **args*, ***kwargs*)

Executes the cmd, with a defined walltime.

Args:

- cmd (string): Command string to execute over the channel
- walltime (int) : Timeout in seconds

Returns:

- (exit_code, stdout, stderr) (int, string, string)

push_file(*source*, *dest_dir*)

Channel will take care of moving the file from source to the destination directory

Args: source (string) : Full filepath of the file to be moved dest_dir (string) : Absolute path of the directory to move to

Returns: destination_path (string)

script_dir
This is a property. Returns the directory assigned for storing all internal scripts such as scheduler submit scripts. This is usually where error logs from the scheduler would reside on the channel destination side.

Args:

- None

Returns:

- Channel script dir

LocalChannel

class libsubmit.channels.local.local.**LocalChannel**(*userhome='.'*, *envs={}*, *scriptDir='./.scripts'*, ***kwargs*)
This is not even really a channel, since opening a local shell is not heavy and done so infrequently that they do not need a persistent channel

__init__(*userhome='.'*, *envs={}*, *scriptDir='./.scripts'*, ***kwargs*)
Initialize the local channel. scriptDir is required by set to a default.

KwArgs:

- userhome (string): (default='.') This is provided as a way to override and set a specific userhome
- envs (dict) : A dictionary of env variables to be set when launching the shell
- channel_script_dir (string): (default="./.scripts") Directory to place scripts

__repr__()
Return repr(self).

close()
There's nothing to close here, and this really doesn't do anything

Returns:

- False, because it really did not "close" this channel.

execute_no_wait(*cmd*, *walltime*)
Synchronously execute a commandline string on the shell.

Args:

- cmd (string) : Commandline string to execute
- walltime (int) : walltime in seconds, this is not really used now.

Returns:

- retcode : Return code from the execution, -1 on fail
- stdout : stdout string
- stderr : stderr string

Raises: None.

execute_wait(*cmd*, *walltime*)
Synchronously execute a commandline string on the shell.

Args:

- cmd (string) : Commandline string to execute

- walltime (int) : walltime in seconds, this is not really used now.

Returns:

- retcode : Return code from the execution, -1 on fail
- stdout : stdout string
- stderr : stderr string

Raises: None.

push_file(*source*, *dest_dir*)

If the source files dirpath is the same as dest_dir, a copy is not necessary, and nothing is done. Else a copy is made.

Args:

- source (string) : Path to the source file
- dest_dir (string) : Path to the directory to which the files is to be copied

Returns:

- destination_path (String) : Absolute path of the destination file

Raises:

- FileCopyException : If file copy failed.

script_dir

This is a property. Returns the directory assigned for storing all internal scripts such as scheduler submit scripts. This is usually where error logs from the scheduler would reside on the channel destination side.

Args:

- None

Returns:

- Channel script dir

SshChannel

class libsubmit.channels.ssh.ssh.**SshChannel**(*hostname*, *username=None*, *password=None*, *scriptDir=None*, ***kwargs*)

Ssh persistent channel. This enables remote execution on sites accessible via ssh. It is assumed that the user has setup host keys so as to ssh to the remote host. Which goes to say that the following test on the commandline should work :

```
>>> ssh <username>@<hostname>
```

__init__(*hostname*, *username=None*, *password=None*, *scriptDir=None*, ***kwargs*)

Initialize a persistent connection to the remote system. We should know at this point whether ssh connectivity is possible

Args:

- hostname (String) : Hostname

KWargs:

- username (string) : Username on remote system
- password (string) : Password for remote system

- channel_script_dir (string) : Full path to a script dir where generated scripts could be sent to.

Raises:

__repr__()
Return repr(self).

__weakref__
list of weak references to the object (if defined)

execute_no_wait(*cmd*, *walltime=2*, *envs={}*)
Execute asynchronously without waiting for exitcode

Args:

- cmd (string): Commandline string to be executed on the remote side
- walltime (int): timeout to exec_command

KWargs:

- envs (dict): A dictionary of env variables

Returns:

- None, stdout (readable stream), stderr (readable stream)

Raises:

- ChannelExecFailed (reason)

pull_file(*remote_source*, *local_dir*)
Transport file on the remote side to a local directory

Args:

- remote_source (string): remote_source
- local_dir (string): Local directory to copy to

Returns:

- str: Local path to file

Raises:

- FileExists : Name collision at local directory.
- FileCopyException : FileCopy failed.

push_file(*local_source*, *remote_dir*)
Transport a local file to a directory on a remote machine

Args:

- local_source (string): Path
- remote_dir (string): Remote path

Returns:

- str: Path to copied file on remote machine

Raises:

- BadScriptPath : if script path on the remote side is bad
- BadPermsScriptPath : You do not have perms to make the channel script dir
- FileCopyException : FileCopy failed.

SshILChannel

class libsubmit.channels.ssh_il.ssh_il.**SshILChannel**(*hostname*, *username=None*, *password=None*, *scriptDir=None*, ***kwargs*)

Ssh persistent channel. This enables remote execution on sites accessible via ssh. It is assumed that the user has setup host keys so as to ssh to the remote host. Which goes to say that the following test on the commandline should work :

```
>>> ssh <username>@<hostname>
```

__init__(*hostname*, *username=None*, *password=None*, *scriptDir=None*, ***kwargs*)

Initialize a persistent connection to the remote system. We should know at this point whether ssh connectivity is possible

Args:

- hostname (String) : Hostname

KWargs:

- username (string) : Username on remote system
- password (string) : Password for remote system
- channel_script_dir (string) : Full path to a script dir where generated scripts could be sent to.

Raises:

6.5.10 Launchers

Launchers are basically wrappers for user submitted scripts as they are submitted to a specific execution resource.

singleNodeLauncher

libsubmit.launchers.**singleNodeLauncher**(*cmd_string*, *taskBlocks*, *walltime=None*)

Worker launcher that wraps the user's cmd_string with the framework to launch multiple cmd_string invocations in parallel. This wrapper sets the bash env variable CORES to the number of cores on the machine. By setting taskBlocks to an integer or to a bash expression the number of invocations of the cmd_string to be launched can be controlled.

Args:

- cmd_string (string): The command string to be launched
- taskBlock (string) : bash evaluated string.

KWargs:

- walltime (int) : This is not used by this launcher.

srunLauncher

libsubmit.launchers.**srunLauncher**(*cmd_string*, *taskBlocks*, *walltime=None*)

Worker launcher that wraps the user's cmd_string with the SRUN launch framework to launch multiple cmd invocations in parallel on a single job allocation.

Args:

- cmd_string (string): The command string to be launched
- taskBlock (string) : bash evaluated string.

KWargs:

- walltime (int) : This is not used by this launcher.

srunMpiLauncher

libsubmit.launchers.**srunMpiLauncher**(*cmd_string*, *taskBlocks*, *walltime=None*)

Worker launcher that wraps the user's cmd_string with the SRUN launch framework to launch multiple cmd invocations in parallel on a single job allocation.

Args:

- cmd_string (string): The command string to be launched
- taskBlock (string) : bash evaluated string.

KWargs:

- walltime (int) : This is not used by this launcher.

6.5.11 Flow Control

This section deals with functionality related to controlling the flow of tasks to various different execution sites.

FlowControl

class parsl.dataflow.flow_control.**FlowControl**(*dfk*, *config*, **args*, *threshold=20*, *interval=5*)

Implements threshold-interval based flow control.

The overall goal is to trap the flow of apps from the workflow, measure it and redirect it the appropriate executors for processing.

This is based on the following logic:

```
BEGIN (INTERVAL, THRESHOLD, callback) :
    start = current_time()

    while (current_time()-start < INTERVAL) :
         count = get_events_since(start)
         if count >= THRESHOLD :
             break

    callback()
```

This logic ensures that the callbacks are activated with a maximum delay of *interval* for systems with infrequent events as well as systems which would generate large bursts of events.

Once a callback is triggered, the callback generally runs a strategy method on the sites available as well asqeuque

TODO: When the debug logs are enabled this module emits duplicate messages. This issue needs more debugging. What I've learnt so far is that the duplicate messages are present only when the timer thread is started, so this could be from a duplicate logger being added by the thread.

close()
Merge the threads and terminate.

make_callback(*kind=None*)
Makes the callback and resets the timer.

KWargs:

- kind (str): Default=None, used to pass information on what triggered the callback

notify(*event_id*)
Let the FlowControl system know that there is an event.

FlowNoControl

class parsl.dataflow.flow_control.**FlowNoControl**(*dfk*, *config*, **args*, *threshold=2*, *interval=2*)
FlowNoControl implements similar interfaces as FlowControl.

Null handlers are used so as to mimic the FlowControl class.

__init__(*dfk*, *config*, **args*, *threshold=2*, *interval=2*)
Initialize the flowcontrol object. This does nothing.

Args:

- dfk (DataFlowKernel) : DFK object to track parsl progress
- config (dict) : Config dict structure

KWargs:

- threshold (int) : Tasks after which the callback is triggered
- interval (int) : seconds after which timer expires

__weakref__
list of weak references to the object (if defined)

close()
This close fn does nothing.

notify(*event_id*)
This notifiy fn does nothing.

Timer

class parsl.dataflow.flow_control.**Timer**(*callback*, **args*, *interval=5*)
This timer is a simplified version of the FlowControl timer. This timer does not employ notify events.

This is based on the following logic :

```
BEGIN (INTERVAL, THRESHOLD, callback) :
    start = current_time()

    while (current_time()-start < INTERVAL) :
         wait()
         break

    callback()
```

__init__(*callback*, **args*, *interval=5*)
Initialize the flowcontrol object We start the timer thread here

Args:

- dfk (DataFlowKernel) : DFK object to track parsl progress
- config (dict) : Config dict structure

KWargs:

- threshold (int) : Tasks after which the callback is triggered
- interval (int) : seconds after which timer expires

__weakref__
list of weak references to the object (if defined)

close()
Merge the threads and terminate.

make_callback(*kind=None*)
Makes the callback and resets the timer.

Strategy

Strategies are responsible for tracking the compute requirements of a workflow as it is executed and scaling the resources to match it.

class parsl.dataflow.strategy.**Strategy**(*dfk*)
FlowControl Strategy.

As a workflow dag is processed by Parsl, new tasks are added and completed asynchronously. Parsl interfaces executors with execution providers to construct scalable execution sites to handle the variable work-load generated by the workflow. This component is responsible for periodically checking outstanding tasks and available compute capacity and trigger scaling events to match workflow needs.

Here's a diagram of a site. A site consists of blocks, which are usually created by single requests to a Local Resource Manager (LRM) such as slurm, condor, torque, or even AWS API. The blocks could contain several task blocks which are separate instances on workers.

```
       |<--minBlocks     |<-initBlocks              maxBlocks-->|
       +-------------------------------------------------------+
       |  +--------Block--------+       +--------Block--------+ |
Site = |  | TaskBlock TaskBlock | ...   | TaskBlock TaskBlock | |
       |  +---------------------+       +---------------------+ |
       +-------------------------------------------------------+
```

The general shape and bounds of a site are user specified through:

1. minBlocks: Minimum number of blocks to maintain per site
2. initBlocks: number of blocks to provision at initialization of workflow
3. maxBlocks: Maximum number of blocks that can be active at a site from one workflow.

```
slots = current_capacity * taskBlocks

active_tasks = pending_tasks + running_tasks

Parallelism = slots / tasks
            = [0, 1] (i.e,  0 <= p <= 1)
```

For example:

When p = 0, => compute with the least resources possible. infinite tasks are stacked per slot.

```
blocks =  minBlocks           { if active_tasks = 0
          max(minBlocks, 1)   {  else
```

When p = 1, => compute with the most resources. one task is stacked per slot.

```
blocks = min ( maxBlocks,
         ceil( active_tasks / slots ) )
```

When p = 1/2, => We stack upto 2 tasks per slot before we overflow and request a new block

let's say min:init:max = 0:0:4 and taskBlocks=2

In the diagram, X <- task

at 2 tasks :

```
+---Block---|
|           |
| X       X |
|slot   slot|
+-----------+
```

at 5 tasks, we overflow as the capacity of a single block is fully used.

```
+---Block---|       +---Block---|
| X       X | ----> |           |
| X       X |       | X         |
|slot   slot|       |slot   slot|
+-----------+       +-----------+
```

__init__(*dfk*)
Initialize strategy.

__weakref__
list of weak references to the object (if defined)

6.5.12 Memoization

class parsl.dataflow.memoization.**Memoizer**(*dfk*, *memoize=True*, *checkpoint={}*)
Memoizer is responsible for ensuring that identical work is not repeated.

When a task is repeated, i.e., the same function is called with the same exact arguments, the result from a previous execution is reused. wiki

The memoizer implementation here does not collapse duplicate calls at call time, but works **only** when the result of a previous call is available at the time the duplicate call is made.

For instance:

```
No advantage from                 Memoization helps
memoization here:                 here:

 TaskA                            TaskB
   |   TaskA                        |
   |     |   TaskA                 done  (TaskB)
```

(continues on next page)

(continued from previous page)

```
   |      |      |                                        (TaskB)
done      |      |
       done      |
              done
```

The memoizer creates a lookup table by hashing the function name and its inputs, and storing the results of the function.

When a task is ready for launch, i.e., all of its arguments have resolved, we add its hash to the task datastructure.

__init__(*dfk*, *memoize=True*, *checkpoint={}*)
Initialize the memoizer.

If either the appCache global config or the memoize kwarg is set to false, memoization is disabled.

Args:

- dfk (DFK obj): The DFK object

KWargs:

- memoize (Bool): enable memoization or not.
- checkpoint (Dict): A checkpoint loaded as a dict.

__weakref__
list of weak references to the object (if defined)

check_memo(*task_id*, *task*)
Create a hash of the task and its inputs and check the lookup table for this hash.

If present, the results are returned. The result is a tuple indicating whether a memo exists and the result, since a Null result is possible and could be confusing. This seems like a reasonable option without relying on an cache_miss exception.

Args:

- task(task) : task from the dfk.tasks table

Returns: Tuple of the following: - present (Bool): Is this present in the memo_lookup_table - Result (Py Obj): Result of the function if present in table

This call will also set task['hashsum'] to the unique hashsum for the func+inputs.

hash_lookup(*hashsum*)
Lookup a hash in the memoization table.

Will raise a KeyError if hash is not in the memoization lookup table.

Args:

- hashsum (str?): The same hashes used to uniquely identify apps+inputs

Returns:

- Lookup result, this is unlikely to be None, since the hashes are set by this library and could not miss entried in it's dict.

Raises:

- KeyError: if hash not in table

make_hash (*task*)
Create a hash of the task inputs.

This uses a serialization library borrowed from ipyparallel. If this fails here, then all ipp calls are also likely to fail due to failure at serialization.

Args:

- task (dict) : Task dictionary from dfk.tasks

Returns:

- hash (str) : A unique hash string

update_memo (*task_id*, *task*, *r*)
Updates the memoization lookup table with the result from a task.

Args:

- task_id (int): Integer task id
- task (dict) : A task dict from dfk.tasks
- r (Result future): Result future

A warning is issued when a hash collision occures during the update. This is not likely.

6.6 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/*
```

- genindex
- modindex
- search

Python Module Index

p

Index

Symbols

T

U