Parsl Documentation

Release 0.9.0

The Parsl Team

Oct 25, 2019

Contents

Parsl is a Python library for programming and executing data-oriented workflows (dataflows) in parallel. Parsl scripts allow selected Python functions and external applications (called apps) to be connected by shared input/output data objects into flexible parallel workflows. Rather than explicitly defining a dependency graph and/or modifying data structures, instead developers simply annotate Python functions and Parsl constructs a dynamic, parallel execution graph derived from the implicit linkage between apps based on shared input/output data objects. Parsl then executes apps when dependencies are met. Parsl is resource-independent, that is, the same Parsl script can be executed on a laptop, cluster, cloud, or supercomputer.

Parsl can be used to realize a variety of workflows:

- Parallel dataflow 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.

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
```

You'll also need gcc to be installed in order to run Parsl.

```
$ gcc --version
```

Parsl has been tested on Linux and MacOS.

Note: As of Parsl v0.7.2 we are switching to an opt-in model for anonymous usage tracking. To help support the Parsl project we request that users opt-in where possible by setting `PARSL_TRACKING=true` in their environment or by setting `usage_tracking=True` in the configuration object (*`parsl.config.Config`*). To read more about what information is collected and how it is used see *Usage statistics collection*.

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 set up 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

or

$ conda config --add channels conda-forge
$ conda install parsl
```

To update a previously installed parsl to a newer version, use: `python3 -m pip install -U parsl`

1.1.3 Installation of Optional Packages

Parsl supports several optional components that require additional module installations. For example support for Amazon Web Services, Extreme Scale Executor etc require additional packages that can be installed easily via `pip` using a pip extras option.

Here's a list of the components and their extras option:

- Amazon Web Services (Cloud) : `aws`
- Google Cloud : `google_cloud`
- Kubernetes : `kubernetes`
- Extreme Scale Executor (Supercomputing) : `extreme_scale`
- Logging monitoring data to a database: `monitoring`
- Jetstream (NSF Cloud) : `jetstream`

Optional extras can be installed using the following syntax:

```
$ python3 -m pip install parsl[<optional_package1>, <optional_package2>]
```

1.2 For Developers

1. Download Parsl:

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

2. Install:

```
$ cd parsl
$ pip install .
( To install specific extra options from the source :)
$ pip install .[<optional_pacakge1>...]
```

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 **@python_app** decorator, and Apps that call external applications using the **@bash_app** decorator. Decorated functions can run in parallel when all their inputs are ready.

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

[ ]:
```
import parsl
import os
from parsl.app.app import python_app, bash_app
from parsl.configs.local_threads import config

#parsl.set_stream_logger() # <-- log everything to stdout

print(parsl.__version__)
```

2.1 Configuring Parsl

Parsl separates code and execution. To do so, it relies on a configuration model to describe the pool of resources to be used for execution (e.g., clusters, clouds, threads).

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

[ ]:
```
parsl.load(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 (`python_app`) and also command-line apps executed via Bash (`bash_app`).

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 **@python_app** decorator.

```
[ ]: @python_app
     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.

```
[ ]: @python_app
     def multiply(a, b):
         return a * b

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

As Parsl apps are potentially executed remotely, they must contain all required dependencies in the function body. For example, if an app requires the time library, it should import that library within the function.

```
[ ]: @python_app
     def slow_hello ():
         import time
         time.sleep(5)
         return 'Hello World!'

     print(slow_hello().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.

As a first example of a Bash app, let's use the Linux command `echo` to return the string 'Hello World!'. This function is made into a Bash App using the **@bash_app** decorator.

Note that the `echo` command will print 'Hello World!' to stdout. In order to use this output, we need to tell Parsl to capture stdout. This is done by specifying the `stdout` keyword argument in the app function. The same approach can be used to capture `stderr`.

```
[ ]: @bash_app
     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())
```

2.4 Passing data

Parsl Apps can exchange data as Python objects (as shown above) or in the form of files. In order to enforce dataflow semantics, Parsl must track the data that is passed into and out of an App. To make Parsl aware of these dependencies, the app function includes `inputs` and `outputs` keyword arguments.

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

```
[ ]: for i in range(3):
         with open(os.path.join(os.getcwd(), 'hello-{}.txt'.format(i)), 'w') as f:
             f.write('hello {}\n'.format(i))
```

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

```
[ ]: @bash_app
     def cat(inputs=[], outputs=[]):
         return 'cat {} > {}'.format(" ".join(inputs), outputs[0])

     concat = cat(inputs=[os.path.join(os.getcwd(), 'hello-0.txt'),
                         os.path.join(os.getcwd(), 'hello-1.txt'),
                         os.path.join(os.getcwd(), 'hello-2.txt')],
                  outputs=[os.path.join(os.getcwd(),'all_hellos.txt')])

     # Open the concatenated file
     with open(concat.outputs[0].result(), 'r') as f:
          print(f.read())
```

2.4.1 Futures

When a normal 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 preferable that functions are executed asynchronously. Parsl provides such asynchronous behavior by returning a future in lieu of results. A future is essentially an object that allows Parsl 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.

2.5 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 of the app and control the workflow.

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

```
[ ]: @python_app
     def hello ():
         import time
         time.sleep(5)
         return 'Hello World!'

```

(continues on next page)

(continued from previous page)

```
app_future = hello()

# Check if the app_future is resolved, which it won't be
print('Done: {}'.format(app_future.done()))

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

2.6 DataFutures

While AppFutures represent the execution of an asynchronous app, DataFutures represent 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.

```
[ ]: # App that echos an input message to an output file
@bash_app
def slowecho(message, outputs=[]):
    return 'sleep 5; echo %s &> %s' % (message, outputs[0])

# Call slowecho specifying the output file
hello = slowecho('Hello World!', outputs=[os.path.join(os.getcwd(), 'hello-world.txt
↪')])

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

# Also check the AppFuture
print('Done: {}'.format(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: {}'.format(hello.done()))
```

2.6.1 Data Management

Parsl is designed to enable implementation of dataflow patterns. These patterns enable workflows, in which the data passed between apps manages the flow of execution, to be defined. 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.

2.7 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.

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

# App that copies the contents of a file to another file
@bash_app
def copy(inputs=[], outputs=[]):
     return 'cat %s &> %s' % (inputs[0], outputs[0])

# Create a test file
open(os.path.join(os.getcwd(), 'cat-in.txt'), 'w').write('Hello World!\n')

# Create Parsl file objects
parsl_infile = File(os.path.join(os.getcwd(), 'cat-in.txt'),)
parsl_outfile = File(os.path.join(os.getcwd(), '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())
```

2.8 Remote Files

Parsl is also able to represent remotely accessible files. In this case, you can instantiate a file object using the remote location of the file. Parsl will implictly stage the file to the execution environment before executing any dependent apps. Parsl will also translate the location of the file into a local file path so that any dependent apps can access the file in the same way as a local file. Parsl supports files that are accessible via Globus, FTP, and HTTP.

Here we create a File object using a publicly accessible file with random numbers. We can pass this file to the `sort_numbers` app in the same way we would a local file.

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

@python_app
def sort_numbers(inputs=[]):
    with open(inputs[0].filepath, 'r') as f:
        strs = [n.strip() for n in f.readlines()]
        strs.sort()
        return strs

unsorted_file = File('https://raw.githubusercontent.com/Parsl/parsl-tutorial/master/
↪input/unsorted.txt')

f = sort_numbers(inputs=[unsorted_file])
print (f.result())
```

2.8.1 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.

2.9 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.

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

# App that writes a variable to a file
@bash_app
def save(variable, outputs=[]):
      return 'echo %s &> %s' % (variable, outputs[0])

# Generate a random number between 1 and 10
random = generate(10)
print('Random number: %s' % random.result())

# Save the random number to a file
saved = save(random, outputs=[os.path.join(os.getcwd(), 'sequential-output.txt')])

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

2.10 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. Note that this takes 5 seconds to run (the time needed for the longest delay), not the 15 seconds that would be needed if these generate functions were called and returned in sequence.

[ ]:
```
# App that generates a random number after a delay
@python_app
def generate(limit,delay):
    from random import randint
    import time
    time.sleep(delay)
    return randint(1,limit)

# Generate 5 random numbers between 1 and 10
rand_nums = []
for i in range(5):
```

(continues on next page)

(continued from previous page)

```
    rand_nums.append(generate(10,i))

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

# Print results
print(outputs)
```

2.11 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. The calls to the first App each create a file, and the second App reads these files and creates a new one. The final App returns the sum as a Python integer.

```
[ ]: # App that generates a semi-random number between 0 and 32,767
@bash_app
def generate(outputs=[]):
    return "echo $(( RANDOM )) &> {outputs[0]}"

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

# App that calculates the sum of values in a list of input files
@python_app
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 semi-random numbers in parallel
output_files = []
for i in range (5):
     output_files.append(generate(outputs=[os.path.join(os.getcwd(), 'random-{}.txt'.
↪format(i))]))

# Concatenate the files into a single file
cc = concat(inputs=[i.outputs[0].filepath for i in output_files],
            outputs=[os.path.join(os.getcwd(), 'all.txt')])

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

2.11.1 Examples

2.12 Monte Carlo workflow

Many scientific applications use the Monte Carlo method to compute results.

One example is calculating π by randomly placing points in a box and using the ratio that are placed inside the circle.

Specifically, if a circle with radius r is inscribed inside a square with side length $2r$, the area of the circle is πr^2 and the area of the square is $(2r)^2$.

Thus, if N uniformly-distributed random points are dropped within the square, approximately $N\pi/4$ 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
```

```
[ ]: # App that estimates pi by placing points in a box
@python_app
def pi(num_points):
    from random import random

    inside = 0
    for i in range(num_points):
        x, y = random(), random()  # Drop a random point in the box.
        if x**2 + y**2 < 1:        # Count points within the circle.
            inside += 1

    return (inside*4 / num_points)

# App that computes the mean of three values
@python_app
def mean(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 mean of the three estimates
mean_pi  = mean(a, b, c)

# Print the results
print("a: {:.5f} b: {:.5f} c: {:.5f}".format(a.result(), b.result(), c.result()))
print("Average: {:.5f}".format(mean_pi.result()))
```

2.12.1 Execution and configuration

Parsl is designed to support arbitrary execution providers (e.g., PCs, clusters, supercomputers, clouds) and execution models (e.g., threads, pilot jobs). 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.

So far in this tutorial, we've used a built-in configuration for running with threads. Below, we will illustrate how to create configs for different environments.

2.13 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 executors that will be used as well as other options such as authentication method (e.g., if using SSH), checkpoint files, and executor specific configuration. In the case of threads we define the maximum number of threads to be used.

```
[ ]: from parsl.config import Config
     from parsl.executors.threads import ThreadPoolExecutor

     local_threads = Config(
         executors=[
             ThreadPoolExecutor(
                 max_threads=8,
                 label='local_threads'
             )
         ]
     )
```

2.14 Local execution with pilot jobs

We can also define a configuration that uses Parsl's HighThroughputExecutor. In this mode, pilot jobs are used to manage the submission. Parsl creates an interchange to manage execution and deploys one or more workers to execute tasks. The following config will instantiate this infrastructure locally, it can be extended to include a remote provider (e.g., the Cori or Theta supercomputers) for execution.

```
[ ]: from parsl.providers import LocalProvider
     from parsl.channels import LocalChannel
     from parsl.config import Config
     from parsl.executors import HighThroughputExecutor

     local_htex = Config(
         executors=[
             HighThroughputExecutor(
                 label="htex_Local",
                 worker_debug=True,
                 cores_per_worker=1,
                 provider=LocalProvider(
                     channel=LocalChannel(),
                     init_blocks=1,
                     max_blocks=1,
```

(continues on next page)

(continued from previous page)

```
            ),
        )
    ],
    strategy=None,
)
```

2.15 Running a workflow using a configuration

We can now run the same workflow using either of the two configurations defined above.

First we clear the current configuration and then load one of the two configurations we defined above. You can change these configurations back and forth to see the same workflow executed using different execution methods. You will notice that executing using the HighThroughputExecutor takes longer, as it has to start interchange/worker processes locally before executing the tasks.

Note: the ‘’parsl-workflows” notebook shows how to execute a Parsl workflow on a remote resource.

[ ]:
```
parsl.clear()
#parsl.load(local_threads)
parsl.load(local_htex)
```

[ ]:
```
@bash_app
def generate(outputs=[]):
    return "echo $(( RANDOM )) &> {outputs[0]}"

@bash_app
def concat(inputs=[], outputs=[]):
    return "cat {0} > {1}".format(" ".join(inputs), outputs[0])

@python_app
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 semi-random numbers
output_files = []
for i in range (5):
     output_files.append(generate(outputs=[os.path.join(os.getcwd(), 'random-%s.txt'
↪% i)]))

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

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

print (total.result())
```

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. Furthermore, the dependency graph is dynamically built and then updated while the Parsl script executes. That is, the graph is not computed in advance and is only complete when the script finishes executing. Apps that have all their dependencies met are slated for execution (in parallel).

The following example demonstrates how a MapReduce job can be defined.

```
from parsl import load, python_app
from parsl.configs.local_threads import config
load(config)

# Map function that returns double the input integer
@python_app
def app_double(x):
    return x*2

# Reduce function that returns the sum of a list
@python_app
def app_sum(inputs=[]):
    return sum(inputs)
```

(continues on next page)

(continued from previous page)

```
# Create a list of integers
items = range(0,10)

# 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 an app decorator. Currently two types of apps can be defined: Python, with the corresponding `@python_app` decorator, and Bash, with the corresponding `@bash_app` decorator. 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 Python function `double(x: int)`, used to double the input value. This function is defined as a Parsl app using the `@python_app` decorator.

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., they cannot include variables used elsewhere in the script).

```
@python_app
def double(x):
      return x * 2

double(x)
```

Python apps may also act upon files. In order to make Parsl aware of these files they must be defined using the `inputs` or `outputs` keyword arguments. The following code snippet illustrates how the contents of one file can be copied to another.

```
@python_app
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 objects to/from functions. Therefore, Python apps can only use input and output objects that can be serialized by cloudpickle or pickle.
5. STDOUT and STDERR produced by Python apps remotely are not captured.

Special Keyword Arguments

Any Parsl app (a Python function decorated with the `@python_app` or `@bash_app` decorator) can use the following special reserved keyword arguments.

1. inputs: (list) This keyword argument defines a list of input *Futures*. Parsl will establish a dependency on these inputs and wait for the results of these futures to be resolved before execution. This is useful if one wishes to pass in an arbitrary number of futures at call time; note that if *Futures* are passed as positional arguments, they will also be resolved before execution.
2. outputs: (list) This keyword argument defines a list of 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.
3. walltime: (int) If the app runs longer than `walltime` seconds, a `parsl.app.errors.AppTimeout` will be raised.

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 future itself holds the python object(s) returned by the app. In case of an error or app failure, the future holds the exception raised by the app.

3.2.2 Bash Apps

Parsl's Bash app is used to wrap the execution of external applications from the command-line. 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 Bash app that will print a message to stdout. Any command-line invocation represented by an arbitrarily long string can be returned by a function decorated within a `@bash_app` to be executed. Unlike Python apps, Bash apps cannot return Python objects, instead they communicate by passing files. The decorated `@bash_app` function provides the same inputs and outputs keyword arguments to manage input and output files. It also includes keyword arguments for capturing the STDOUT and STDERR streams and recording them in files that are managed by Parsl.

```
@bash_app
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
```

(continues on next page)

(continued from previous page)

```
# 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 or parsl.AUTO_LOGNAME) The path to a file to which standard output should be redirected. If set to `parsl.AUTO_LOGNAME`, the log will be automatically named according to task id and saved under `task_logs` in the run directory.
4. stderr: (string or parsl.AUTO_LOGNAME) The path to a file to which standard error should be redirected. If set to `parsl.AUTO_LOGNAME`, the log will be automatically named according to task id and saved under `task_logs` in the run directory.
5. label: (string) If the app is invoked with `stdout=parsl.AUTO_LOGNAME` or `stderr=parsl.AUTO_LOGNAME`, append `label` to the log name.

A Bash app allows for the composition of the string to execute on the command-line from the arguments passed to the decorated function. The string that is returned is formatted by the Python string format (PEP 3101).

```
@bash_app
def echo(arg, inputs=[], stderr=parsl.AUTO_LOGNAME, stdout=parsl.AUTO_LOGNAME):
    return 'echo {} {} {}'.format(arg, inputs[0], inputs[1])

future = echo('Hello', inputs=['World', '!'])
future.result() # block until task has completed

with open(future.stdout, 'r') as f:
    print(f.read()) # prints "Hello World !"
```

Returns

A Bash app returns an AppFuture just like a Python app; however the value returned inside the AppFuture has no real meaning.

If a bash app exits with unix exit code 0, then the AppFuture will complete. If a bash app exits with any other code, this will be treated as a failure, and the AppFuture will instead contain an AppFailure exception. The unix edit code can be accessed through the `exitcode` attribute of that AppFailure.

3.3 Futures

When a Python function is invoked, the Python interpreter waits for the function to complete execution and returns the results. When functions execute for a long period of time it may not be desirable to wait for completion, instead it is often preferable that the function executes asynchronously. Parsl provides such asynchronous behavior by returning a future in lieu of results. A future is essentially an object that can be used 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 may not yet be available.

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

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 Python's concurrent library. AppFutures provide several key functionalities:

1. An AppFuture provides a way to check the current status of an app.

```
@python_app
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 an app:

```
@python_app
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.

```
@python_app
def bad_divide(x):
    return 6/x

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

# 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 by a specific app, 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 fail.

3.3.2 DataFutures

While AppFutures represent the execution of an asynchronous app, DataFutures represent the files an app produces. Parsl's dataflow model, in which data is passed from one app to another via files, requires such a construct to enable apps to validate the 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. DataFutures are accessible through the `outputs` attribute of the AppFuture. DataFutures are inherited from Python's concurrent library.

The following code snippet shows how DataFutures are used:

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

# 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().filepath, 'r') as f:
     print(f.read())
```

Note: Adding `.filepath` is only needed on python 3.5. With python >= 3.6 the resulting file can maybe be passed to open directly.

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 implementation of a wide range of workflow patterns from sequential through to complex nested, parallel workflows.

Parsl is also designed to address broad execution requirements from workflows that run a large number of very small tasks 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 that this example demonstrates the use of both Python and Bash apps.

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

# Write a message to a file
@bash_app
def save(message, outputs=[]):
      return 'echo {} &> {}'.format(message, outputs[0])

message = generate(10)

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

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.

```
@python_app
def wait_sleep_double(x, foo_1, foo_2):
     import time
     time.sleep(2)   # Sleep for 2 seconds
     return x*2

# Launch two apps, which will execute in parallel, since they do not 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

One of the most common ways that Parsl apps are executed in parallel is via loops. The following example shows how a simple loop can be used to create many random numbers in parallel.

```
@python_app
def generate(limit):
    from random import randint
```

(continues on next page)

(continued from previous page)

```
    """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 the following example a set of files, each with a random number, is created by the generate app. These files are then concatenated into a single file, which is subsequently used to compute the sum of all numbers.

```
@bash_app
def generate(outputs=[]):
    return 'echo $(( RANDOM % (10 - 5 + 1 ) + 5 )) &> {}'.format(outputs[0])

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

@python_app
def total(inputs=[]):
    total = 0
    with open(inputs[0].filepath, '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, opportunities for concurrent execution.

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 flexible 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 FTP, HTTP, HTTPS, and Globus.

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 URL *scheme*: Local, FTP, HTTP, HTTPS and Globus.

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. The following example executes the `cat` command on a local file:

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

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

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

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

FTP, HTTP, HTTPS

File objects with FTP, HTTP, and HTTPS schemes represent remote files on FTP, HTTP and HTTPS servers, respectively.The following example defines a file accessible on a remote FTP server.

```
File('ftp://www.iana.org/pub/mirror/rirstats/arin/ARIN-STATS-FORMAT-CHANGE.txt')
```

When such a file object is passed as an input to an app, Parsl will download the file to the executor where the app is scheduled for execution. The following example illustrates how the remote file is implicitly downloaded from an FTP server and then converted. Note: the app does not need to know the local location of the downloaded file as Parsl abstracts this translation.

```
@python_app
def convert(inputs=[], outputs=[]):
    with open(inputs[0].filepath, 'r') as inp:
        content = inp.read()
        with open(outputs[0].filepath, 'w') as out:
            out.write(content.upper())

# create an remote Parsl file
inp = File('ftp://www.iana.org/pub/mirror/rirstats/arin/ARIN-STATS-FORMAT-CHANGE.txt')
```

(continues on next page)

(continued from previous page)

```
# create a local Parsl file
out = File('file:///tmp/ARIN-STATS-FORMAT-CHANGE.txt')

# call the convert app with the Parsl file
f = convert(inputs=[inp], outputs=[out])
f.result()
```

Globus

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.

When Globus files are passed as inputs or outputs to/from an app, Parsl stage the files to/from the remote executor using Globus. The staging occurs implicitly. That is, Parsl is responsible for transferring the input file from the Globus endpoint to the executor, or transferring the output file from the executor to the Globus endpoint. 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.

```
@python_app
def sort_strings(inputs=[], outputs=[]):
    with open(inputs[0].filepath, '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')

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

Configuration

In order to manage where data is staged users may configure the default `working_dir` on a remote executor. This is passed to the `ParslExecutor` via the `working_dir` parameter in the *Config* instance. For example:

```
from parsl.config import Config
from parsl.executors.ipp import IPyParallelExecutor

config = Config(
    executors=[
        IPyParallelExecutor(
            working_dir="/home/user/parsl_script"
        )
```

(continues on next page)

(continued from previous page)

```
    ]
)
```

When using the Globus scheme Parsl requires knowledge of the Globus endpoint that is associated with an executor. This is done by specifying the `endpoint_name` (the UUID of the Globus endpoint that is associated with the system) in the configuration.

In some cases, for example when using a Globus shared endpoint or when a Globus endpoint 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.

```
from parsl.config import Config
from parsl.executors.ipp import IPyParallelExecutor
from parsl.data_provider.globus import GlobusStaging
from parsl.data_provider.data_manager import default_staging

config = Config(
    executors=[
        IPyParallelExecutor(
            working_dir="/home/user/parsl_script",
            storage_access=default_staging + [GlobusStaging(
                endpoint_uuid="7d2dc622-2edb-11e8-b8be-0ac6873fc732",
                endpoint_path="/",
                local_path="/home/user"
            )]
        )
    ]
)
```

Authorization

In order to interact with Globus, you must be authorised. The first time that you use Globus with Parsl, prompts will take you through an authorization procedure involving your web browser. You can authorize without having to run a script (for example, if you're running your script in a batch system where it will be unattended) by running this command line:

```
$ parsl-globus-auth
Parsl Globus command-line authoriser
If authorisation to Globus is necessary, the library will prompt you now.
Otherwise it will do nothing
Authorization complete
```

rsync

`rsync` can be used to transfer files in the `file:` scheme in configurations where workers cannot access the submit side filesystem directly, such as when executing on an AWS EC2 instance.

Configuration

rsync must be installed on both the submit and worker side. It can usually be installed using the operating system package manager - for example `apt-get install rsync`.

The parameter to RSyncStaging should describe the prefix to be passed to each rsync command to connect from workers to the submit side host. This will often be the username and public IP address of the submitting system.

```
from parsl.data_provider.rsync import RSyncStaging

config = Config(
    executors=[
        HighThroughputExecutor(
            storage_access=[HTTPInTaskStaging(), FTPInTaskStaging(), RSyncStaging(
↪"benc@" + public_ip)],
            ...
    )
)
```

Authorization

The rsync staging provider delegates all authentication and authorization to the underlying rsync command. This command must be correctly authorized to connect back to the submitting system. The form of this authorization will depend on the systems in question.

This example installs an ssh key from the submit side filesystem and turns off host key checking, in the worker_init initialization of an EC2 instance. The ssh key must have sufficient privileges to run rsync over ssh on the submitting system.

```
with open("rsync-callback-ssh", "r") as f:
    private_key = f.read()

ssh_init = """
mkdir .ssh
chmod go-rwx .ssh

cat > .ssh/id_rsa <<EOF
{private_key}
EOF

cat > .ssh/config <<EOF
Host *
  StrictHostKeyChecking no
EOF

chmod go-rwx .ssh/id_rsa
chmod go-rwx .ssh/config

""".format(private_key=private_key)

config = Config(
    executors=[
        HighThroughputExecutor(
            storage_access=[HTTPInTaskStaging(), FTPInTaskStaging(), RSyncStaging(
↪"benc@" + public_ip)],
            provider=AWSProvider(
            ...
            worker_init = ssh_init
            ...
            )
```

(continues on next page)

(continued from previous page)

```
    )
)
```

3.6 Execution

Parsl scripts can be executed on different execution providers (e.g., PCs, clusters, supercomputers) and using different execution models (e.g., threads, pilot jobs, etc.). Parsl separates the code from the configuration that specifies which execution provider(s) and executor(s) to use. Parsl provides a high level abstraction, called a *block*, for providing a uniform description of a resource configuration irrespective of the specific execution provider.

Note: Refer to *Configuration* for information on how to configure the various subsystems described below for your workflow's resource requirements.

3.6.1 Execution providers

Execution providers are responsible for managing execution resources. In the simplest case a PC could be used for execution. 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, HTCondor 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 currently supports the following providers:

1. `LocalProvider`: The provider allows you to run locally on your laptop or workstation.
2. `CobaltProvider`: This provider allows you to schedule resources via the Cobalt scheduler.
3. `SlurmProvider`: This provider allows you to schedule resources via the Slurm scheduler.
4. `CondorProvider`: This provider allows you to schedule resources via the Condor scheduler.
5. `GridEngineProvider`: This provider allows you to schedule resources via the GridEngine scheduler.
6. `TorqueProvider`: This provider allows you to schedule resources via the Torque scheduler.
7. `AWSProvider`: This provider allows you to provision and manage cloud nodes from Amazon Web Services.
8. `GoogleCloudProvider`: This provider allows you to provision and manage cloud nodes from Google Cloud.
9. `JetstreamProvider`: This provider allows you to provision and manage cloud nodes from Jetstream (NSF Cloud).
10. `KubernetesProvider`: This provider allows you to provision and manage containers on a Kubernetes cluster.
11. `AdHocProvider`: This provider allows you manage execution over a collection of nodes to form an ad-hoc cluster.
12. `LSFProvider`: This provider allows you to schedule resources via IBM's LSF scheduler

3.6.2 Executors

Depending on the execution provider there are a number of ways to execute workloads on that resource. For example, for local execution a thread pool could be used, for supercomputers pilot jobs or various launchers could 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 resource requirements of the workflow.

Parsl currently supports the following executors:

1. *`ThreadPoolExecutor`*: This executor supports multi-thread execution on local resources.
2. *`HighThroughputExecutor`*: The HighThroughputExecutor is designed as a replacement for the IPyParallelExecutor. Implementing hierarchical scheduling and batching, the HighThroughputExecutor consistently delivers high throughput task execution on the order of 1000 Nodes
3. *`WorkQueueExecutor`*: The WorkQueueExecutor integrates Work Queue as an execution backend. Work Queue scales to tens of thousands of cores and implements reliable execution of tasks with dynamic resource sizing.
4. *`IPyParallelExecutor`* [**Deprecated**]: 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.
5. *`ExtremeScaleExecutor`*: [**Beta**] The ExtremeScaleExecutor uses `mpi4py` to scale over 4000+ nodes. This executor is typically used for executing on Supercomputers.
6. `Swift/TurbineExecutor`: [**Deprecated**] 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.

Note: Refer to *Configuration* for information on how to configure these executors.

3.6.3 Launchers

On many traditional batch systems, the user is expected to request a large number of nodes and launch tasks using a system such as srun (for slurm), aprun (for crays), mpirun etc. Launchers are responsible for abstracting these different task-launch systems to start the appropriate number of workers across cores and nodes. Parsl currently supports the following set of launchers:

1. *`SrunLauncher`*: Srun based launcher for Slurm based systems.
2. *`AprunLauncher`*: Aprun based launcher for Crays.
3. *`SrunMPILauncher`*: Launcher for launching MPI applications with Srun.
4. *`GnuParallelLauncher`*: Launcher using GNU parallel to launch workers across nodes and cores.
5. *`MpiExecLauncher`*: Uses Mpiexec to launch.
6. *`SimpleLauncher`*: The launcher default to a single worker launch.
7. *`SingleNodeLauncher`*: This launcher launches `workers_per_node` count workers on a single node.

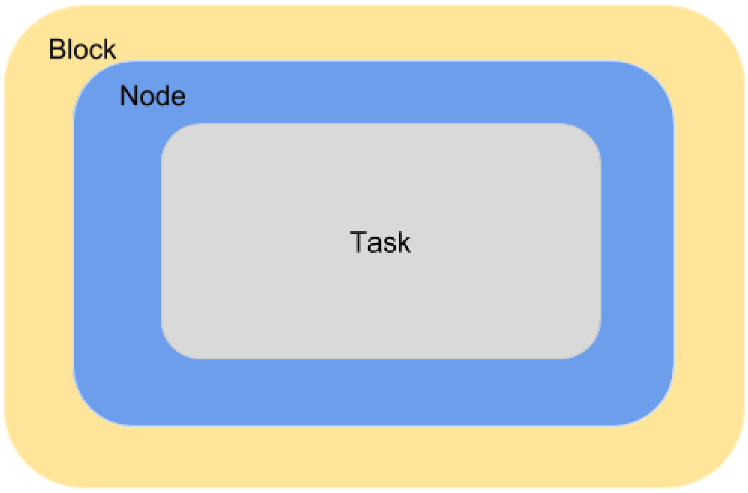
Additionally, custom launchers which are aware of more specific environments (for example, to launch node processes inside containers with custom environments) can be written as part of the workflow configuration. For example, this launcher uses Srun to launch `worker-wrapper`, passing the command to be run as parameters to worker-wrapper. It is the responsibility of worker-wrapper to launch the command it is given inside the appropriate environment.

```
class MyShifterSRunLauncher:
    def __init__(self):
        self.srun_launcher = SrunLauncher()

    def __call__(self, command, tasks_per_node, nodes_per_block):
        new_command="worker-wrapper {}".format(command)
        return self.srun_launcher(new_command, tasks_per_node, nodes_per_block)
```

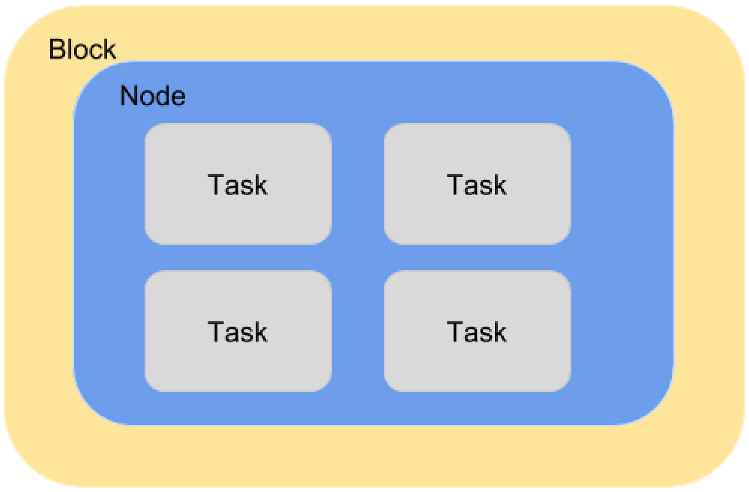
3.6.4 Blocks

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 execute *tasks* (instances of apps) within and across (e.g., for MPI jobs) nodes. Three different examples of block configurations are shown below.

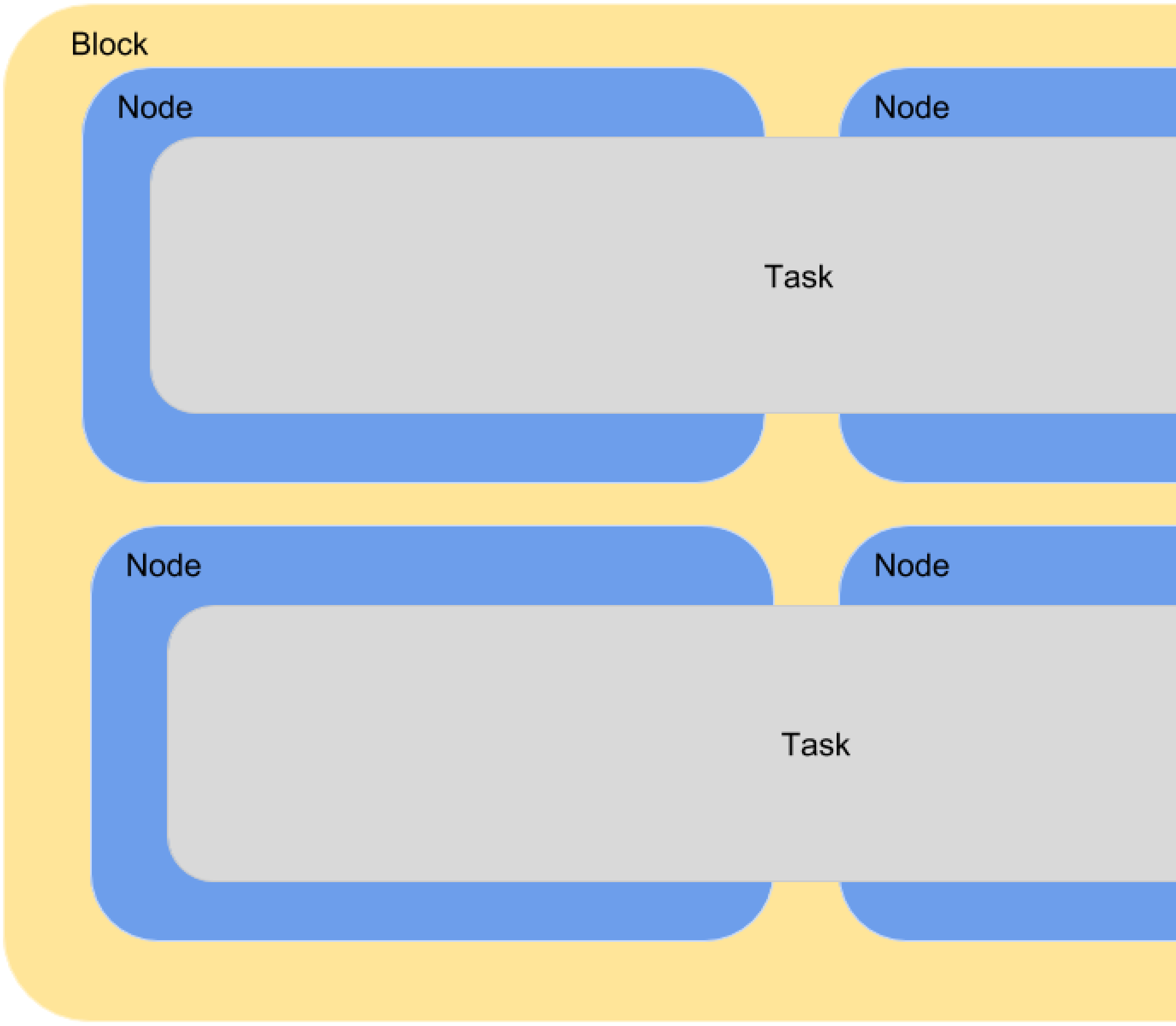
1. A single block comprised of a node executing one task:



2. A single block comprised on a node executing several tasks. This configuration is most suitable for single threaded apps running on multicore target systems. The number of tasks executed concurrently is proportional to the number of cores available on the system.



3. A block comprised of several nodes and executing several tasks, where a task can span multiple nodes. This configuration is generally used by MPI applications. Starting a task requires using a specific MPI launcher that is supported on the target system (e.g., aprun, srun, mpirun, mpiexec).



3.6.5 Elasticity

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.

During execution Parsl 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 can 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 an executor. The script configuration for an executor defines the minimum, maximum, and initial number of blocks to be used.

The configuration options for specifying elasticity bounds are:

1. `min_blocks`: Minimum number of blocks to maintain per executor.
2. `init_blocks`: Initial number of blocks to provision at initialization of workflow.
3. `max_blocks`: Maximum number of blocks that can be active per executor.

The configuration options for specifying the shape of each block are:

1. `workers_per_node`: Number of workers started per node, which corresponds to the number of tasks that can execute concurrently on a node.
2. `nodes_per_block`: Number of nodes requested per block.

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 executor as well as a parameter (*p*) to control the level of parallelism. Parallelism is expressed as the ratio of task execution capacity and the sum of running tasks and available tasks (tasks with their dependencies met, but waiting for execution). 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., min_blocks) are used. By selecting a fraction between 0 and 1, the aggressiveness in provisioning resources can be controlled.

For example:

- When p = 0: Use the fewest resources possible.

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

- When p = 1: Use as many resources as possible.

```
blocks = min(max_blocks,
             ceil((running_tasks + available_tasks) / (workers_per_node * nodes_per_
↪block))
```

- When p = 1/2: Stack up to 2 tasks 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 2 tasks. Parallelism of 0.5 means that when more than 2 * the total task capacity are queued a new block will be requested (up to 2 possible blocks). An example *`Config`* is:

```
from parsl.config import Config
from libsubmit.providers.local.local import Local
from parsl.executors.ipp import IPyParallelExecutor

config = Config(
    executors=[
        IPyParallelExecutor(
```

(continues on next page)

(continued from previous page)

```
                label='local_ipp',
                workers_per_node=2,
                provider=Local(
                    min_blocks=1,
                    init_blocks=1,
                    max_blocks=4,
                    nodes_per_block=1,
                    parallelism=0.5
                )
            )
        ]
)
```

The animated diagram below illustrates the behavior of this executor. In the diagram, the tasks are allocated to the first block, until 5 tasks are submitted. At this stage, as more than double the available task capacity is used, Parsl provisions a new block for executing the remaining tasks.

3.6.6 Multi-executor

Parsl supports the definition of any number of executors in the configuration, as well as specifying which of these executors can 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 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.

The following code snippet shows how executors can be specified in the app decorator.

```
#(CPU heavy app) (CPU heavy app) (CPU heavy app) <--- Run on compute queue
#      |                 |                |
#    (data)            (data)           (data)
#       \                |               /
#       (Analysis and visualization phase)         <--- Run on GPU node

# A mock molecular dynamics simulation app
@bash_app(executors=["Theta.Phi"])
def MD_Sim(arg, outputs=[]):
    return "MD_simulate {} -o {}".format(arg, outputs[0])

# Visualize results from the mock MD simulation app
@bash_app(executors=["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

Parsl provides support for capturing, tracking, and handling a variety of errors. It also provides functionality to appropriately respond to failures during workflow execution. If a task is unable to complete execution within a specified time limit or if it is unable to produce the specified set of outputs it is considered to have failed.

Failures might occur for various reasons:

1. Task exceeded 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. Task failed to launch, for example if an input dependency is not met.

Since Parsl tasks are executed asynchronously, it can be difficult to determine 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:

```
@python_app
def bad_divide(x):
    return 6 / x

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

# Catch and handle the exception.
try:
     doubled_x.result()
except ZeroDivisionError as e:
     print('Oops! You tried to divide by 0.')
except Exception as e:
     print('Oops! Something really bad happened.')
```

3.7.2 Retries

Often errors in distributed/parallel environments are transient. Retrying a task is a common method for adding resiliency to a workflow. By retrying failed apps, transient failures (e.g., machine failure, network failure) and intermittent failures within applications can be overcome. When `retries` are enabled (and set to an integer > 0), Parsl will automatically re-launch applications that have failed, until the retry limit is reached.

By default `retries = 0`.

The following example shows how the number of retries can be set to 2:

```
from parsl import load
from parsl.tests.configs.local_threads import config
config.retries = 2

load(config)
```

3.7.3 Lazy fail

Warning: Due to a known bug (issue#282), disabling lazy_errors with `lazy_errors=False` is **not** supported in Parsl 0.6.0.

Parsl implements a lazy failure model through which a workload will continue to execute in the case that some tasks fail. That is, the workflow does not halt as soon as it encounters a failure, but continues execution of every app that is unaffected.

For example:

```
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 ----->
```

3.8 App caching

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 re-executed, wasting valuable developer time and computation resources. App caching solves this problem by storing results from apps that have completed so that they can be re-used. App caching can be enabled by setting the `cache` argument in the `python_app()` or `bash_app()` decorator to `True` (by default it is `False`). App caching can be globally disabled by setting `app_cache=False` in the *`Config`*.

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

App caching 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 during development. Using app caching will ensure that only modified apps are re-executed.

3.8.1 Caveats

It is important to consider several important issues when using app caching:

- Determinism: App caching is generally useful only when the apps are deterministic. If the outputs may be different for identical inputs, app caching 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 app caching 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.

3.9 Checkpointing

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

Note: Checkpointing is *only* possible for apps which have AppCaching enabled. If AppCaching is disabled in the config `Config.app_cache`, checkpointing will **not** work.

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

When loading a checkpoint file the Parsl script will use checkpointed results for any apps that have been previously executed. Like app caching, 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.

Parsl provides four checkpointing modes:

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

```
>>> from parsl.configs.local_threads import config
>>> config.checkpoint_mode = 'task_exit'
```

2. `periodic`: a checkpoint is created periodically using a user-specified checkpointing interval.

```
>>> from parsl.configs.local_threads import config
>>> config.checkpoint_mode = 'periodic'
>>> config.checkpoint_period = "01:00:00"
```

3. `dfk_exit`: checkpoints are created when Parsl is about to exit. This reduces the risk of losing results due to premature workflow termination from exceptions, terminate signals, etc. However it is still possible that information might be lost if the workflow is terminated abruptly (machine failure, SIGKILL, etc.)

```
>>> from parsl.configs.local_threads import config
>>> config.checkpoint_mode = '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.

```
>>> import parsl
>>> from parsl.configs.local_threads import config
>>> dfk = parsl.load(config)
>>> ....
>>> dfk.checkpoint()
```

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

3.9.1 Creating a checkpoint

When using automated checkpointing there is no need to modify a Parsl script as checkpointing will be conducted transparently. The following example shows how manual checkpointing can be invoked in a Parsl script.

```
import parsl
from parsl import python_app
from parsl.configs.local_threads import config

dfk = parsl.load(config)

@python_app(cache=True)
def slow_double(x, sleep_dur=1):
    import time
    time.sleep(sleep_dur)
    return x * 2

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 Resuming from a checkpoint

When resuming a workflow from a checkpoint Parsl allows the user to select which checkpoint file(s) to be used. 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:

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

config.checkpoint_files = get_all_checkpoints()

parsl.load(config)
```

3.10 Configuration

Parsl workflows are developed completely independently from their execution environment. There are very many different execution environments in which Parsl programs and their apps can run, and many of these environments have multiple options of how those Parsl programs and apps run, which makes configuration somewhat complex, and also makes determining how to set up Parsl's configuration for a particular set of choices fairly complex, though we think the actual configuration itself is reasonable simple.

Parsl offers an extensible configuration model through which the execution environment and communication within that environment is configured. Parsl is configured using `Config` object. For more information, see the `Config` class documentation. The following shows how the configuration can be specified.

```python
import parsl
from parsl.config import Config
from parsl.executors.threads import ThreadPoolExecutor

config = Config(
    executors=[ThreadPoolExecutor()],
    lazy_errors=True
)
parsl.load(config)
```

Configuration How-To and Examples:

Note: Please note that all configuration examples below require customization for your account, allocation, Python environment, etc.

3.10.1 How to Configure

The configuration provided to Parsl tells Parsl what resources to use to run the Parsl program and apps, and how to use them. Therefore it is important to carefully evaluate certain aspects of the Parsl program and apps, and the planned compute resources, to determine an ideal configuration match. These aspects are: 1) where the Parsl apps will execute;

2) how many nodes will be used to execute the apps, and how long the apps will run; 3) should the scheduler allocate multiple nodes at one time; and 4) where will the main parsl program run and how will it communicate with the apps.

Stepping through the following question should help you formulate a suitable configuration. In addition, examples for some specific configurations follow.

1. Where would you like the apps in the Parsl program to run?

Target	Executor	Provider
Laptop/Workstation	• *ThreadPoolExecutor* • *IPyParallelExecutor* • *HighThroughputExecutor* • *ExtremeScaleExecutor*	*LocalProvider*
Amazon Web Services	• *IPyParallelExecutor* • *HighThroughputExecutor*	*AWSProvider*
Google Cloud	• *IPyParallelExecutor* • *HighThroughputExecutor*	*GoogleCloudProvider*
Slurm based cluster or supercomputer	• *IPyParallelExecutor* • *HighThroughputExecutor* • *ExtremeScaleExecutor*	*SlurmProvider*
Torque/PBS based cluster or supercomputer	• *IPyParallelExecutor* • *HighThroughputExecutor* • *ExtremeScaleExecutor*	*TorqueProvider*
Cobalt based cluster or supercomputer	• *IPyParallelExecutor* • *HighThroughputExecutor* • *ExtremeScaleExecutor*	*CobaltProvider*
GridEngine based cluster or grid	• *IPyParallelExecutor* • *HighThroughputExecutor*	*GridEngineProvider*
Condor based cluster or grid	• *IPyParallelExecutor* • *HighThroughputExecutor*	*CondorProvider*
Kubernetes cluster	• *IPyParallelExecutor* • *HighThroughputExecutor*	*KubernetesProvider*

2. How many nodes will you use to run them? What task durations give good performance on different executors?

Executor	Number of Nodes*[0]	Task duration for good performance
*ThreadPoolExecutor	1 (Only local)	Any
LowLatencyExecutor	<=10	10ms+
IPyParallelExecutor	<=128	50ms+
HighThroughputExecutor	<=2000	Task duration(s)/#nodes >= 0.01 longer tasks needed at higher scale
ExtremeScaleExecutor	>1000, <=8000†[0]	>minutes

Warning: IPyParallelExecutor will be deprecated as of Parsl v0.8.0, with HighThroughputExecutor as the recommended replacement.

3. If you are running on a cluster or supercomputer, will you request multiple nodes per batch (scheduler) job? (Here we use the term block to be equivalent to a batch job.)

nodes_per_block = 1		
Provider	Executor choice	Suitable Launchers
Systems that don't use Aprun	Any	• SingleNodeLauncher • SimpleLauncher
Aprun based systems	Any	• AprunLauncher

nodes_per_block > 1		
Provider	Executor choice	Suitable Launchers
TorqueProvider	Any	• AprunLauncher • MpiExecLauncher
CobaltProvider	Any	• AprunLauncher
SlurmProvider	Any	• SrunLauncher if native slurm • AprunLauncher, otherwise

Note: If you are on a Cray system, you most likely need the AprunLauncher to launch workers unless you are on a **native Slurm** system like *Cori (NERSC)*

4. Where will you run the main Parsl program, given that you already have determined where the apps will run? (This is needed to determine how to communicate between the Parsl program and the apps.)

[0] We assume that each node has 32 workers. If there are fewer workers launched per node, a higher number of nodes could be supported.
[0] 8000 nodes with 32 workers each totalling 256000 workers is the maximum scale at which we've tested the ExtremeScaleExecutor.

Parsl program location	App execution target	Suitable channel
Laptop/Workstation	Laptop/Workstation	*LocalChannel*
Laptop/Workstation	Cloud Resources	None
Laptop/Workstation	Clusters with no 2FA	*SSHChannel*
Laptop/Workstation	Clusters with 2FA	*SSHInteractiveLoginChannel*
Login node	Cluster/Supercomputer	*LocalChannel*

3.10.2 Comet (SDSC)

The following snippet shows an example configuration for executing remotely on San Diego Supercomputer Center's **Comet** supercomputer. The example is designed to be executed on the login nodes, using the `SlurmProvider` to interface with the Slurm scheduler used by Comet and the `SrunLauncher` to launch workers.

Warning: This config has **NOT** been tested with Parsl v0.9.0

```python
from parsl.config import Config
from parsl.launchers import SrunLauncher
from parsl.providers import SlurmProvider
from parsl.executors import HighThroughputExecutor
from parsl.addresses import address_by_query


config = Config(
    executors=[
        HighThroughputExecutor(
            label='Comet_HTEX_multinode',
            address=address_by_query(),
            worker_logdir_root='YOUR_LOGDIR_ON_COMET',
            max_workers=2,
            provider=SlurmProvider(
                'debug',
                launcher=SrunLauncher(),
                # string to prepend to #SBATCH blocks in the submit
                # script to the scheduler
                scheduler_options='',
                # Command to be run before starting a worker, such as:
                # 'module load Anaconda; source activate parsl_env'.
                worker_init='',
                walltime='00:10:00',
                init_blocks=1,
                max_blocks=1,
                nodes_per_block=2,
```

(continues on next page)

(continued from previous page)

```
            ),
        )
    ]
)
```

3.10.3 Cori (NERSC)

The following snippet shows an example configuration for accessing NERSC's **Cori** supercomputer. This example uses the *HighThroughputExecutor* and connects to Cori's Slurm scheduler. 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.

```
from parsl.config import Config
from parsl.providers import SlurmProvider
from parsl.launchers import SrunLauncher
from parsl.executors import HighThroughputExecutor
from parsl.addresses import address_by_interface


config = Config(
    executors=[
        HighThroughputExecutor(
            label='Cori_HTEX_multinode',
            # This is the network interface on the login node to
            # which compute nodes can communicate
            address=address_by_interface('bond0.144'),
            cores_per_worker=2,
            provider=SlurmProvider(
                'regular',  # Partition / QOS
                nodes_per_block=2,
                init_blocks=1,
                # string to prepend to #SBATCH blocks in the submit
                # script to the scheduler eg: '#SBATCH --constraint=knl,quad,cache'
                scheduler_options='',
                # Command to be run before starting a worker, such as:
                # 'module load Anaconda; source activate parsl_env'.
                worker_init='',
                # We request all hyperthreads on a node.
                launcher=SrunLauncher(overrides='-c 272'),
                walltime='00:10:00',
                # Slurm scheduler on Cori can be slow at times,
                # increase the command timeouts
                cmd_timeout=120,
            ),
        )
    ]
)
```

3.10.4 Stampede2 (TACC)

The following snippet shows an example configuration for accessing TACC's **Stampede2** supercomputer. This example uses theHighThroughput executor and connects to Stampede2's Slurm scheduler.

```
from parsl.config import Config
from parsl.providers import SlurmProvider
from parsl.launchers import SrunLauncher
from parsl.executors import HighThroughputExecutor
from parsl.addresses import address_by_hostname
from parsl.data_provider.globus import GlobusStaging


config = Config(
    executors=[
        HighThroughputExecutor(
            label='Stampede2_HTEX',
            address=address_by_hostname(),
            max_workers=2,
            provider=SlurmProvider(
                nodes_per_block=2,
                init_blocks=1,
                min_blocks=1,
                max_blocks=1,
                partition='YOUR_PARTITION',
                # string to prepend to #SBATCH blocks in the submit
                # script to the scheduler eg: '#SBATCH --constraint=knl,quad,cache'
                scheduler_options='',
```

(continues on next page)

(continued from previous page)

```
                # Command to be run before starting a worker, such as:
                # 'module load Anaconda; source activate parsl_env'.
                worker_init='',
                launcher=SrunLauncher(),
                walltime='00:30:00'
            ),
            storage_access=[GlobusStaging(
                endpoint_uuid='ceea5ca0-89a9-11e7-a97f-22000a92523b',
                endpoint_path='/',
                local_path='/'
            )]
        )

    ],
)
```

3.10.5 Frontera (TACC)

Deployed in June 2019, Frontera is the 5th most powerful supercomputer in the world. Frontera replaces the NSF Blue Waters system at NCSA and is the first deployment in the National Science Foundation's petascale computing program. The configuration below assumes that the user is running on a login node and uses the *SlurmProvider* to interface with the scheduler, and uses the *SrunLauncher* to launch workers.

```
from parsl.config import Config
from parsl.channels import LocalChannel
```

(continues on next page)

(continued from previous page)

```
from parsl.providers import SlurmProvider
from parsl.executors import HighThroughputExecutor
from parsl.launchers import SrunLauncher
from parsl.addresses import address_by_hostname


""" This config assumes that it is used to launch parsl tasks from the login nodes
of Frontera at TACC. Each job submitted to the scheduler will request 2 nodes for 10␣
↪minutes.
"""
config = Config(
    executors=[
        HighThroughputExecutor(
            label="frontera_htex",
            address=address_by_hostname(),
            max_workers=1,          # Set number of workers per node
            provider=SlurmProvider(
                cmd_timeout=60,     # Add extra time for slow scheduler responses
                channel=LocalChannel(),
                nodes_per_block=2,
                init_blocks=1,
                min_blocks=1,
                max_blocks=1,
                partition='normal',                                 # Replace with␣
↪partition name
                scheduler_options='#SBATCH -A <YOUR_ALLOCATION>',   # Enter scheduler_
↪options if needed

                # Command to be run before starting a worker, such as:
                # 'module load Anaconda; source activate parsl_env'.
                worker_init='',

                # Ideally we set the walltime to the longest supported walltime.
                walltime='00:10:00',
                launcher=SrunLauncher(),
            ),
        )
    ],
)
```

3.10.6 Theta (ALCF)

The following snippet shows an example configuration for executing on Argonne Leadership Computing Facility's **Theta** supercomputer. This example uses the *HighThroughputExecutor* and connects to Theta's Cobalt scheduler using the *CobaltProvider*. This configuration assumes that the script is being executed on the login nodes of Theta.

```
from parsl.config import Config
from parsl.providers import CobaltProvider
from parsl.launchers import AprunLauncher
```

(continues on next page)

(continued from previous page)

```
from parsl.executors import HighThroughputExecutor
from parsl.addresses import address_by_hostname


config = Config(
    executors=[
        HighThroughputExecutor(
            label='theta_local_htex_multinode',
            max_workers=4,
            address=address_by_hostname(),
            provider=CobaltProvider(
                queue='YOUR_QUEUE',
                account='YOUR_ACCOUNT',
                launcher=AprunLauncher(overrides="-d 64"),
                walltime='00:30:00',
                nodes_per_block=2,
                init_blocks=1,
                min_blocks=1,
                max_blocks=1,
                # string to prepend to #COBALT blocks in the submit
                # script to the scheduler eg: '#COBALT -t 50'
                scheduler_options='',
                # Command to be run before starting a worker, such as:
                # 'module load Anaconda; source activate parsl_env'.
                worker_init='',
                cmd_timeout=120,
            ),
        )
    ],
)
```

3.10.7 Cooley (ALCF)

The following snippet shows an example configuration for executing on Argonne Leadership Computing Facility's **Cooley** analysis and visualization system. The example uses the *HighThroughputExecutor* and connects to Cooley's Cobalt scheduler using the *CobaltProvider*. This configuration assumes that the script is being executed on the login nodes of Theta.

```
from parsl.config import Config
from parsl.executors import HighThroughputExecutor
from parsl.addresses import address_by_hostname
from parsl.launchers import MpiRunLauncher
from parsl.providers import CobaltProvider


config = Config(
    executors=[
        HighThroughputExecutor(
            label="cooley_htex",
            worker_debug=False,
            cores_per_worker=1,
            address=address_by_hostname(),
            provider=CobaltProvider(
                queue='debug',
                account='YOUR_ACCOUNT',  # project name to submit the job
```

(continues on next page)

(continued from previous page)

```
                launcher=MpiRunLauncher(),
                scheduler_options='',  # string to prepend to #COBALT blocks in the␣
↪submit script to the scheduler
                worker_init='',  # command to run before starting a worker, such as
↪'source activate env'
                init_blocks=1,
                max_blocks=1,
                min_blocks=1,
                nodes_per_block=4,
                cmd_timeout=60,
                walltime='00:10:00',
            ),
        )
    ],

)
```

3.10.8 Blue Waters (Cray)

The following snippet shows an example configuration for executing remotely on Blue Waters, a flagship machine at the National Center for Supercomputing Applications. The configuration assumes the user is running on a login node and uses the `TorqueProvider` to interface with the scheduler, and uses the `AprunLauncher` to launch workers.

```
from parsl.config import Config
from parsl.executors import HighThroughputExecutor
from parsl.addresses import address_by_hostname
from parsl.launchers import AprunLauncher
from parsl.providers import TorqueProvider


config = Config(
    executors=[
        HighThroughputExecutor(
            label="bw_htex",
            cores_per_worker=1,
            worker_debug=False,
            address=address_by_hostname(),
            provider=TorqueProvider(
                queue='normal',
                launcher=AprunLauncher(overrides="-b -- bwpy-environ --"),
                scheduler_options='',  # string to prepend to #SBATCH blocks in the␣
↪submit script to the scheduler
```

(continues on next page)

(continued from previous page)

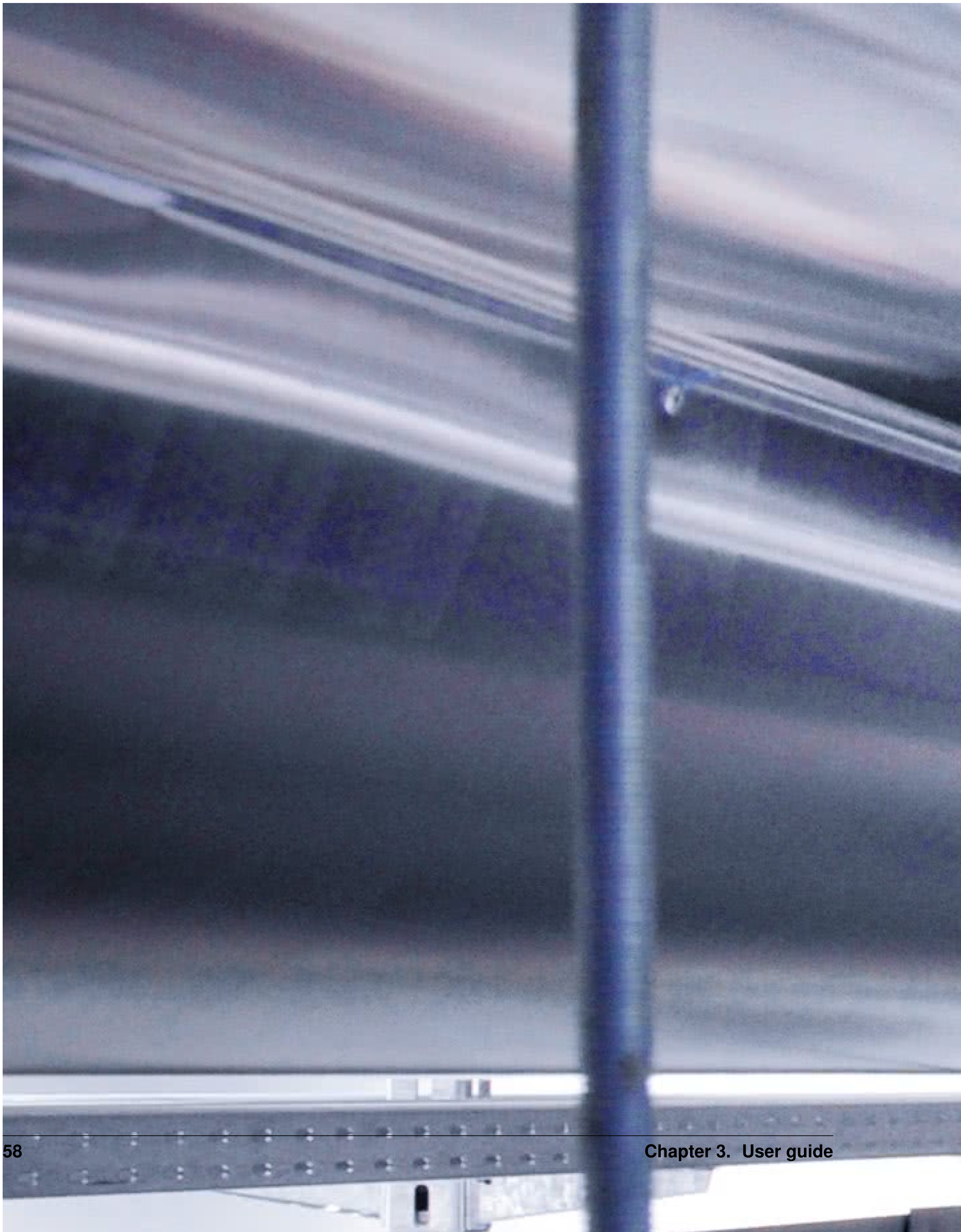
```
                worker_init='',  # command to run before starting a worker, such as
↪'source activate env'
                init_blocks=1,
                max_blocks=1,
                min_blocks=1,
                nodes_per_block=2,
                walltime='00:10:00'
            ),
        )

    ],

)
```

3.10.9 Summit (ORNL)

The following snippet shows an example configuration for executing from the login node on Summit, the leadership class supercomputer hosted at the Oak Ridge National Laboratory. The example uses the `LSFProvider` to provision compute nodes from the LSF cluster scheduler and the `JsrunLauncher` to launch workers across the compute nodes.

```
from parsl.config import Config
from parsl.executors import HighThroughputExecutor

from parsl.launchers import JsrunLauncher
from parsl.providers import LSFProvider

from parsl.addresses import address_by_interface

config = Config(
    executors=[
        HighThroughputExecutor(
            label='Summit_HTEX',
            # On Summit ensure that the working dir is writeable from the compute␣
↪nodes,
            # for eg. paths below /gpfs/alpine/world-shared/
            working_dir='YOUR_WORKING_DIR_ON_SHARED_FS',
            address=address_by_interface('ib0'),  # This assumes Parsl is running on␣
↪login node
            worker_port_range=(50000, 55000),
            provider=LSFProvider(
                launcher=JsrunLauncher(),
                walltime="00:10:00",
                nodes_per_block=2,
                init_blocks=1,
                max_blocks=1,
                worker_init='',  # Input your worker environment initialization␣
↪commands
                project='YOUR_PROJECT_ALLOCATION',
                cmd_timeout=60
            ),
        )

    ],
)
```

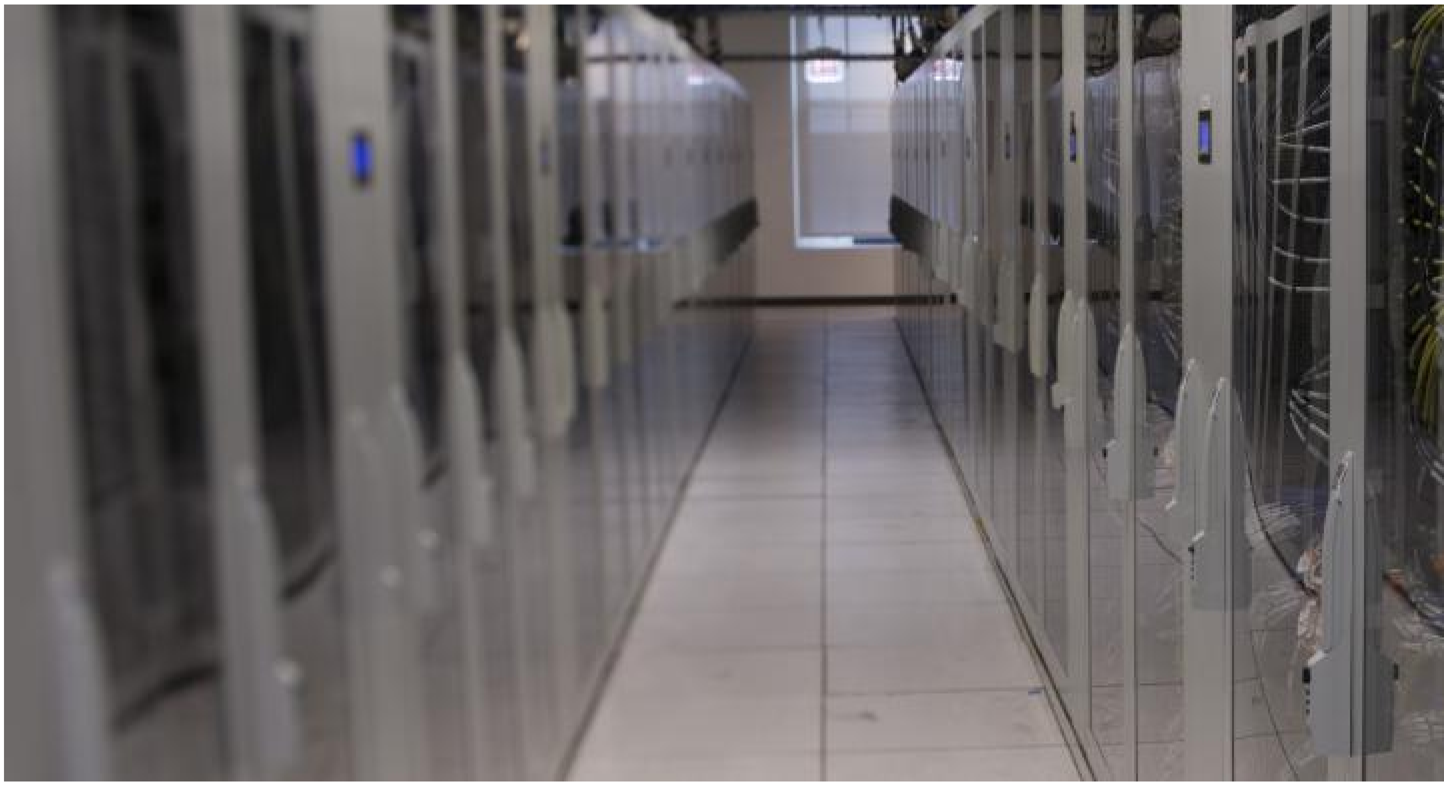
3.10.10 CC-IN2P3



The snippet below shows an example configuration for executing from a login node on IN2P3's Computing Centre. The configuration uses the *LocalProvider* to run on a login node primarily to avoid GSISSH, which Parsl does not support yet. This system uses Grid Engine which Parsl interfaces with using the *GridEngineProvider*.

```
from parsl.config import Config
from parsl.channels import LocalChannel
from parsl.providers import GridEngineProvider
from parsl.executors import HighThroughputExecutor
from parsl.addresses import address_by_query

config = Config(
    executors=[
        HighThroughputExecutor(
            label='cc_in2p3_htex',
            address=address_by_query(),
            max_workers=2,
            provider=GridEngineProvider(
                channel=LocalChannel(),
                nodes_per_block=1,
                init_blocks=2,
                max_blocks=2,
                walltime="00:20:00",
                scheduler_options='',     # Input your scheduler_options if needed
                worker_init='',     # Input your worker_init if needed
            ),
        )
    ],
)
```

3.10.11 Midway (RCC, UChicago)



This Midway cluster is a campus cluster hosted by the Research Computing Center at the University of Chicago. The snippet below shows an example configuration for executing remotely on Midway. The configuration assumes the user is running on a login node and uses the `SlurmProvider` to interface with the scheduler, and uses the `SrunLauncher` to launch workers.

```
from parsl.config import Config
from parsl.providers import SlurmProvider
from parsl.launchers import SrunLauncher
from parsl.addresses import address_by_hostname
from parsl.executors import HighThroughputExecutor

config = Config(
    executors=[
        HighThroughputExecutor(
            label='Midway_HTEX_multinode',
            worker_debug=False,
            address=address_by_hostname(),
            max_workers=2,
            provider=SlurmProvider(
                'YOUR_PARTITION',  # Partition name, e.g 'broadwl'
                launcher=SrunLauncher(),
                nodes_per_block=2,
                init_blocks=1,
                min_blocks=1,
                max_blocks=1,
                # string to prepend to #SBATCH blocks in the submit
                # script to the scheduler eg: '#SBATCH --constraint=knl,quad,cache'
                scheduler_options='',
```

(continues on next page)

(continued from previous page)

```
                # Command to be run before starting a worker, such as:
                # 'module load Anaconda; source activate parsl_env'.
                worker_init='',
                walltime='00:30:00'
            ),
        )
    ],
)
```

3.10.12 Open Science Grid

The Open Science Grid (OSG) is a national, distributed computing Grid spanning over 100 individual sites to provide tens of thousands of CPU cores. The snippet below shows an example configuration for executing remotely on OSG. The configuration uses the `CondorProvider` to interface with the scheduler.

Note: This config was last tested with 0.8.0

```
from parsl.config import Config
from parsl.providers import CondorProvider
from parsl.executors import HighThroughputExecutor
from parsl.addresses import address_by_query

config = Config(
    executors=[
        HighThroughputExecutor(
            label='OSG_HTEX',
            address=address_by_query(),
            max_workers=1,
            provider=CondorProvider(
                nodes_per_block=1,
                init_blocks=4,
                max_blocks=4,
                # This scheduler option string ensures that the compute nodes␣
↪provisioned
                # will have modules
                scheduler_options='Requirements = OSGVO_OS_STRING == "RHEL 6" && Arch␣
↪== "X86_64" &&  HAS_MODULES == True',
                # Command to be run before starting a worker, such as:
                # 'module load Anaconda; source activate parsl_env'.
                worker_init='',
                walltime="00:20:00",
            ),
        )
    ]
)
```

3.10.13 Amazon Web Services

Note: Please note that **boto3** library is a requirement to use AWS with Parsl. This can be installed via `python3 -m pip install parsl[aws]`

Amazon Web Services is a commercial cloud service which allows you to rent a range of computers and other computing services. The snippet below shows an example configuration for provisioning nodes from the Elastic Compute Cloud (EC2) service. The first run would configure a Virtual Private Cloud and other networking and security infrastructure that will be re-used in subsequent runs. The configuration uses the *AWSProvider* to connect to AWS.

```
from parsl.config import Config
from parsl.providers import AWSProvider
from parsl.executors import HighThroughputExecutor
from parsl.addresses import address_by_query

config = Config(
    executors=[
        HighThroughputExecutor(
            label='ec2_single_node',
            address=address_by_query(),
            provider=AWSProvider(
                # Specify your EC2 AMI id
                'YOUR_AMI_ID',
                # Specify the AWS region to provision from
                # eg. us-east-1
                region='YOUR_AWS_REGION',

                # Specify the name of the key to allow ssh access to nodes
```

(continues on next page)

(continued from previous page)

```
                key_name='YOUR_KEY_NAME',
                profile="default",
                state_file='awsproviderstate.json',
                nodes_per_block=1,
                init_blocks=1,
                max_blocks=1,
                min_blocks=0,
                walltime='01:00:00',
            ),
        )
    ],
)
```

3.10.14 Kubernetes Clusters

Kubernetes is an open-source system for container management, such as automating deployment and scaling of containers. The snippet below shows an example configuration for deploying pods as workers on a Kubernetes cluster. The KubernetesProvider exploits the Python Kubernetes API, which assumes that you have kube config in `~/.kube/config`.

```
from parsl.config import Config
from parsl.executors import HighThroughputExecutor
from parsl.providers import KubernetesProvider
from parsl.addresses import address_by_route


config = Config(
    executors=[
        HighThroughputExecutor(
            label='kube-htex',
            cores_per_worker=1,
            max_workers=1,
            worker_logdir_root='YOUR_WORK_DIR',

            # Address for the pod worker to connect back
            address=address_by_route(),
            provider=KubernetesProvider(
                namespace="default",

                # Docker image url to use for pods
                image='YOUR_DOCKER_URL',
```

(continues on next page)

(continued from previous page)

```
                # Command to be run upon pod start, such as:
                # 'module load Anaconda; source activate parsl_env'.
                # or 'pip install parsl'
                worker_init='',

                # The secret key to download the image
                secret="YOUR_KUBE_SECRET",

                # Should follow the Kubernetes naming rules
                pod_name='YOUR-POD-Name',

                nodes_per_block=1,
                init_blocks=1,
                # Maximum number of pods to scale up
                max_blocks=10,
            ),
        ),
    ]
)
```

3.10.15 Ad-Hoc Clusters

Any collection of compute nodes without a scheduler setup for task scheduling can be considered an ad-hoc cluster. Often these machines have a shared filesystem such as NFS or Lustre. In order to use these resources with Parsl, they need to set-up for password-less SSH access.

To use these ssh-accessible collection of nodes as an ad-hoc cluster, we create an executor for each node, using the *LocalProvider* with *SSHChannel* to identify the node by hostname. An example configuration follows.

```
from parsl.providers import AdHocProvider
from parsl.channels import SSHChannel
from parsl.executors import HighThroughputExecutor
from parsl.addresses import address_by_query
from parsl.config import Config

user_opts = {'adhoc':
             {'username': 'YOUR_USERNAME',
              'script_dir': 'YOUR_SCRIPT_DIR',
              'remote_hostnames': ['REMOTE_HOST_URL_1', 'REMOTE_HOST_URL_2']
             }
}

config = Config(
    executors=[
        HighThroughputExecutor(
            label='remote_htex',
            max_workers=2,
            address=address_by_query(),
            worker_logdir_root=user_opts['adhoc']['script_dir'],
            provider=AdHocProvider(
                # Command to be run before starting a worker, such as:
                # 'module load Anaconda; source activate parsl_env'.
                worker_init='',
                channels=[SSHChannel(hostname=m,
```

(continues on next page)

(continued from previous page)

```
                                     username=user_opts['adhoc']['username'],
                                     script_dir=user_opts['adhoc']['script_dir'],
                ) for m in user_opts['adhoc']['remote_hostnames']]
            )
        )
    ],
    #  AdHoc Clusters should not be setup with scaling strategy.
    strategy=None,
)
```

Note: Multiple blocks should not be assigned to each node when using the *HighThroughputExecutor*

Note: Load-balancing will not work properly with this approach. In future work, a dedicated provider that supports load-balancing will be implemented. You can follow progress on this work here.

3.10.16 Further help

For help constructing a configuration, you can click on class names such as *Config* or *HighThroughputExecutor* to see the associated class documentation. The same documentation can be accessed interactively at the python command line via, for example:

```
>>> from parsl.config import Config
>>> help(Config)
```

3.11 Modularizing Parsl workflows

Parsl workflows can be developed in many ways. When developing a simple workflow it is often convenient to include the app definitions and control logic in a single script. However, as a workflow inevitably grows and changes, like any code, there are significant benefits to be obtained by modularizing the workflow, including:

1. Better readability
2. Logical separation of components (e.g., apps, config, and control logic)
3. Ease of reuse of components

Note: Support for isolating configuration loading and app definition is available since 0.6.0. Refer: Issue#50

The following example illustrates how a Parsl project can be organized into modules.

The configuration(s) can be defined in a module or file (e.g., `config.py`) which can be imported into the control script depending on which execution resources should be used.

```
import parsl
from parsl.config import Config
from parsl.executors.threads import ThreadPoolExecutor

local_threads = Config(
```

(continues on next page)

(continued from previous page)

```
  executors=[ThreadPoolExecutor(max_threads=4)],
  lazy_errors=True
)
```

Parsl apps can be defined in separate file(s) or module(s) (e.g., `library.py`) grouped by functionality.

```
from parsl import python_app

@python_app
def increment(x):
    return x + 1
```

Finally, the control logic for the Parsl application can then be implemented in a separate file (e.g., `run_increment.py`). This file must the import the configuration from `config.py` before calling the `increment` app from `library.py`:

```
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 Monitoring

Parsl aims to make the task of running parallel workflows easy by providing monitoring and diagnostic capabilities to help track the state of your workflow, down to the individual applications being executed on remote machines. To enable Parsl's monitoring feature for your workflow, you will need a few additional packages.

3.12.1 Installation

Parsl's monitoring model relies on writing workflow progress to a sqlite database and using separate tools that query this database to create a web-based dashboard for the workflow.

3.12.2 Monitoring configuration

Here's an example configuration that logs monitoring information to a local sqlite database:

```
import parsl
from parsl.monitoring.monitoring import MonitoringHub
from parsl.config import Config
from parsl.executors import HighThroughputExecutor
from parsl.addresses import address_by_hostname

import logging

config = Config(
   executors=[
       HighThroughputExecutor(
           label="local_htex",
           cores_per_worker=1,
           max_workers=4,
           address=address_by_hostname(),
       )
   ],
   monitoring=MonitoringHub(
       hub_address=address_by_hostname(),
       hub_port=55055,
       monitoring_debug=False,
       resource_monitoring_interval=10,
   ),
   strategy=None
)
```

3.12.3 Visualization

Run the `parsl-visualize` utility:

```
$ parsl-visualize
```

If your monitoring database is not the default of `monitoring.db` in the current working directory, you can specify a different database URI on the command line. For example, if the full path to your `monitoring.db` is `/tmp/monitoring.db`, run:

```
$ parsl-visualize sqlite:////tmp/monitoring.db
```

By default, the visualization web server listens on `127.0.0.1:8080`. If you are running on a machine with a web browser, you can access viz_server in the browser via `127.0.0.1:8080`. If you are running on the login node of a cluster, to access viz_server in a local machine's browser, you can use an ssh tunnel from your local machine to the cluster:

```
$ ssh -L 50000:127.0.0.1:8080 username@cluster_address
```

This binds your local machine's port 50000 to the remote cluster's port 8080. This allows you to access viz_server directly on your local machine's browser via `127.0.0.1:50000`.

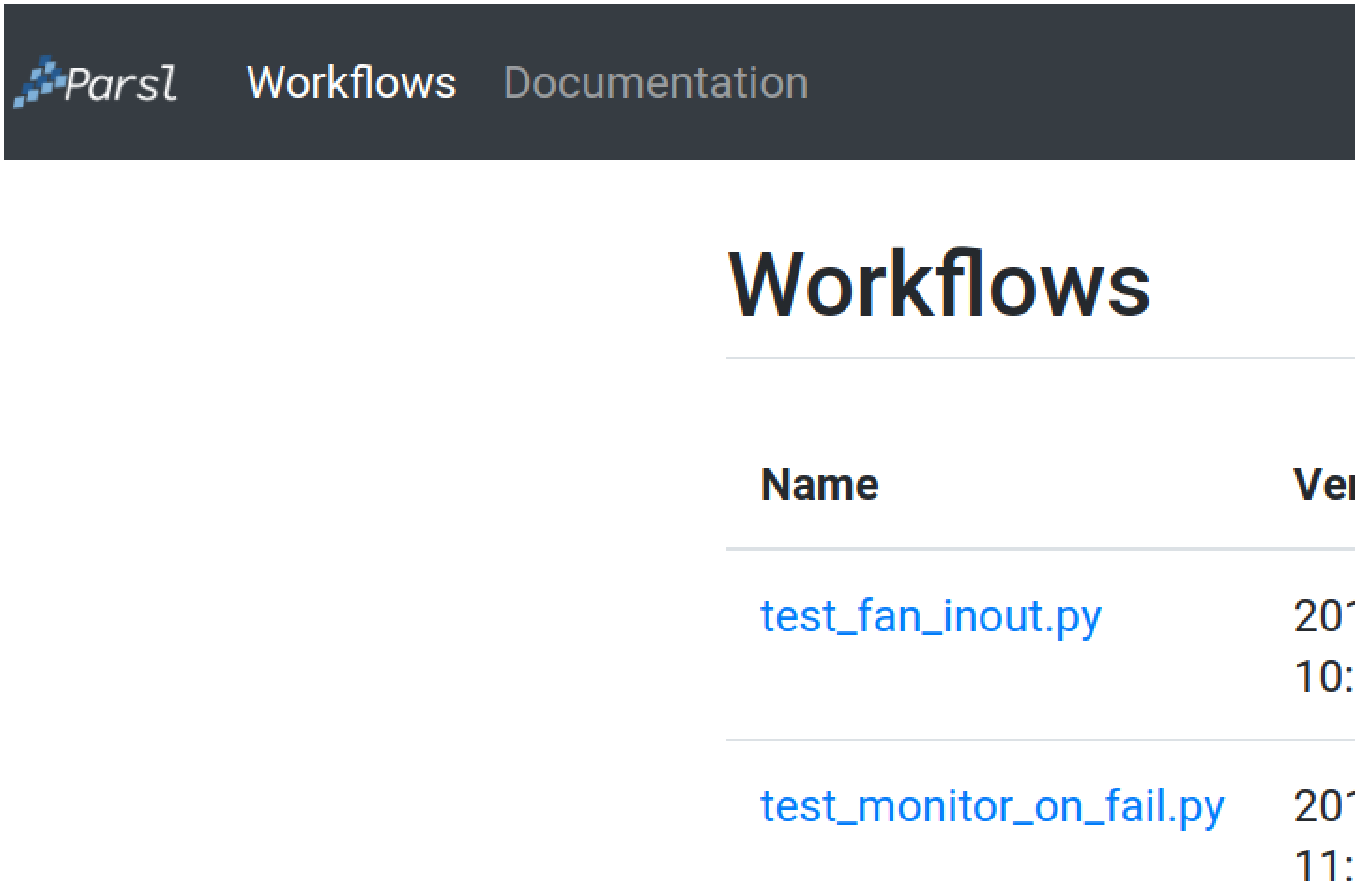
Warning: Below is an alternative to host the viz_server on a cluster, which may violate the cluster's security policy. Please check with your cluster admin before doing this.

If the cluster allows you to host a web server on its public IP address with a specific port (i.e., open to Internet via `public_IP:55555`), you can run:

```
$ parsl-visualize --listen 0.0.0.0 --port 55555
```

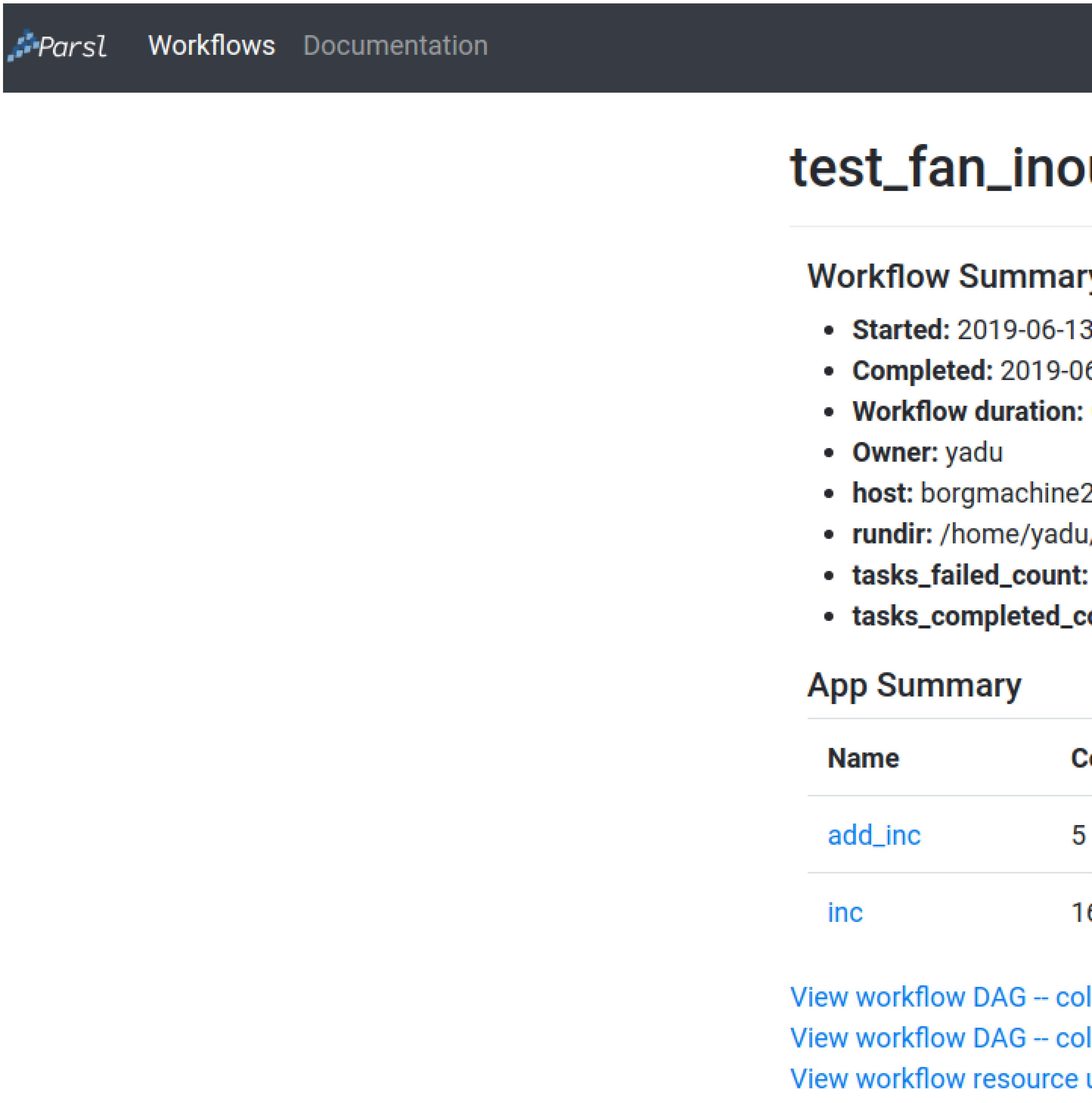
Workflows Page

The workflows page lists all instances of a Parsl workflow that has been executed with monitoring turned on. It also gives a high level overview of workflow runs as a table as shown below:



Throughout the visualization pages, all blue elements are clickable. For eg, clicking a specific worklow name from the table takes you to the Workflow Summary page described in the next section.

Workflow Summary



test_fan_ino

Workflow Summar

- **Started:** 2019-06-13
- **Completed:** 2019-06
- **Workflow duration:**
- **Owner:** yadu
- **host:** borgmachine2
- **rundir:** /home/yadu/
- **tasks_failed_count:**
- **tasks_completed_c**

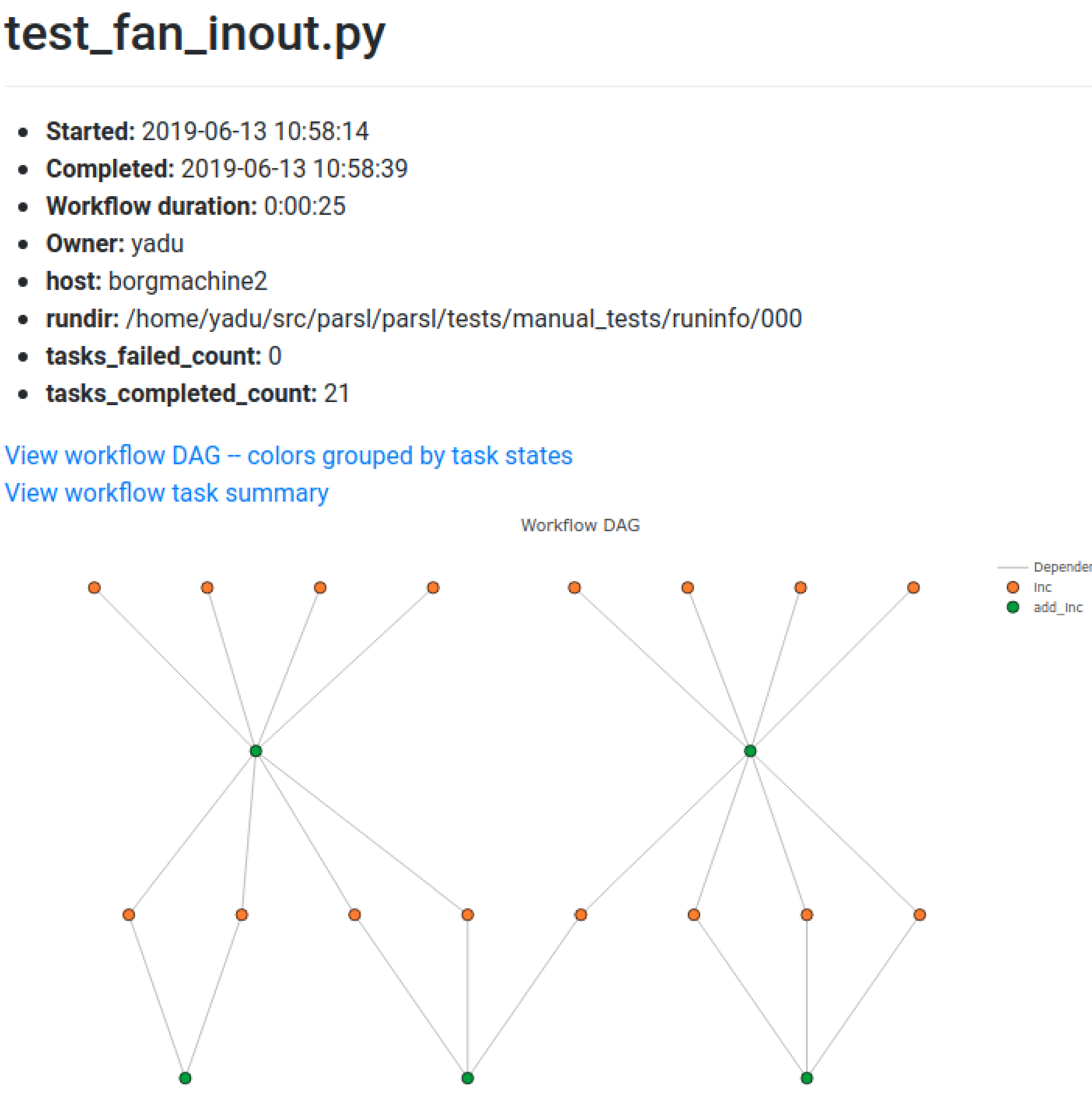
App Summary

Name	C
add_inc	5
inc	1

View workflow DAG -- col
View workflow DAG -- col
View workflow resource

The above screenshot of the workflow summary page captures the run level details such as start and end times as well as task summary statistics. The workflow summary section is followed by the *App Summary* that lists the various apps and count of invocations each. This is followed by three different views of the workflow:

- Workflow DAG - colors grouped by apps: This visualization is useful to visually inspect the dependency structure of the workflow DAG. Hovering over the nodes in the DAG shows a tooltip for the app that the node represents and it's task ID.

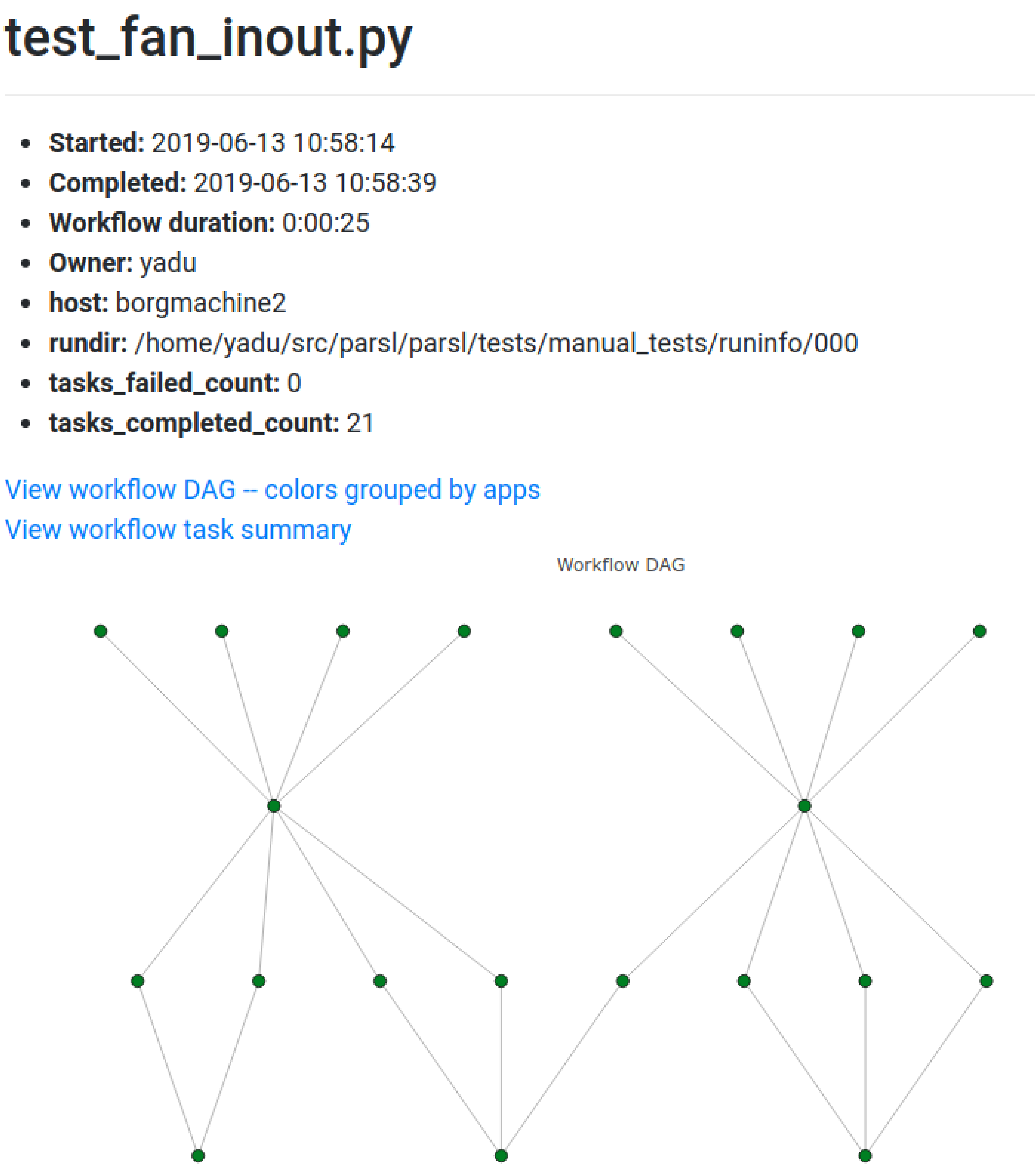


test_fan_inout.py

- **Started:** 2019-06-13 10:58:14
- **Completed:** 2019-06-13 10:58:39
- **Workflow duration:** 0:00:25
- **Owner:** yadu
- **host:** borgmachine2
- **rundir:** /home/yadu/src/parsl/parsl/tests/manual_tests/runinfo/000
- **tasks_failed_count:** 0
- **tasks_completed_count:** 21

View workflow DAG -- colors grouped by task states
View workflow task summary

- Workflow DAG - colors grouped by task states: This visualization is useful to identify what stages in the workflow are complete and what stages are pending.

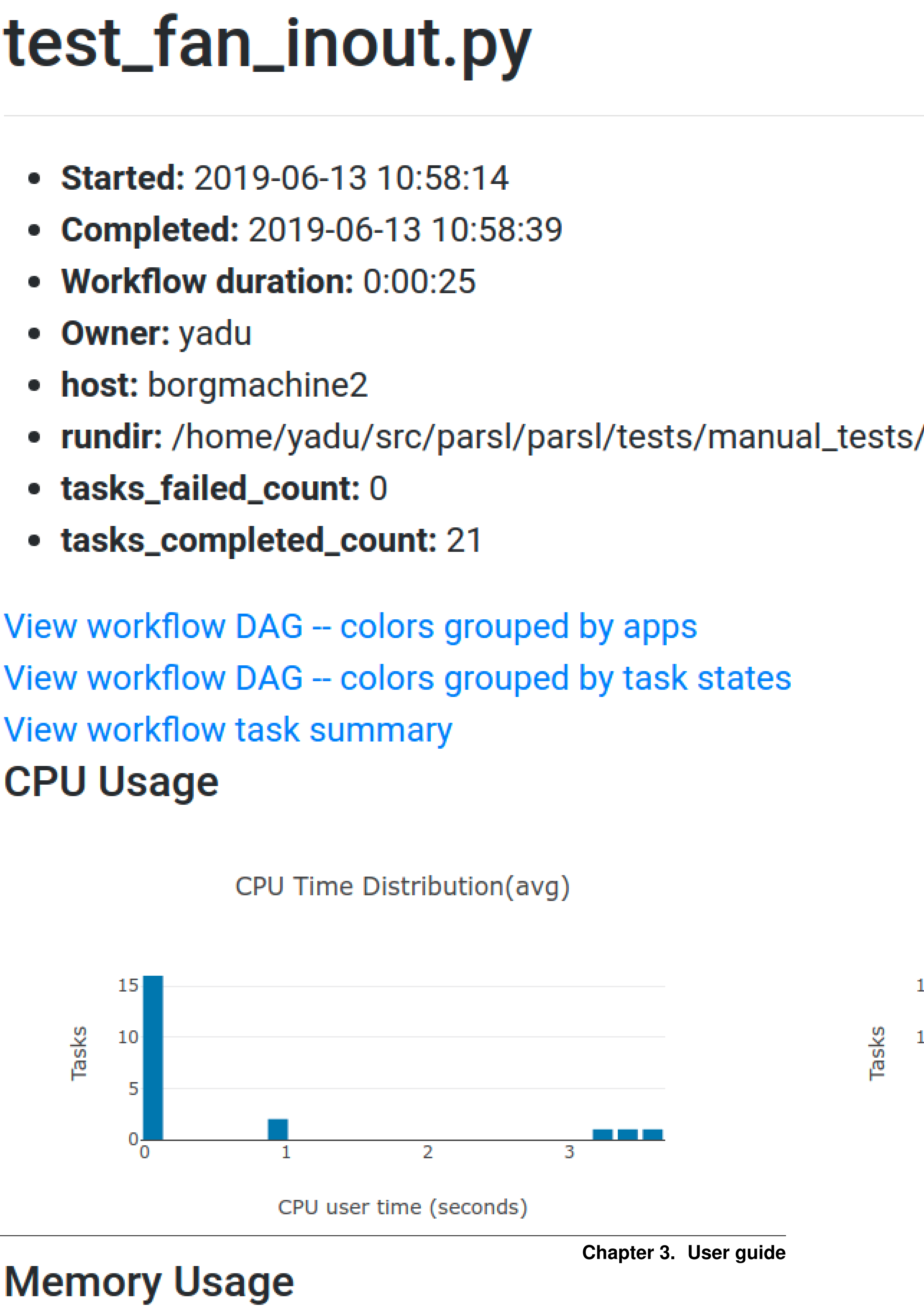


test_fan_inout.py

- **Started:** 2019-06-13 10:58:14
- **Completed:** 2019-06-13 10:58:39
- **Workflow duration:** 0:00:25
- **Owner:** yadu
- **host:** borgmachine2
- **rundir:** /home/yadu/src/parsl/parsl/tests/manual_tests/runinfo/000
- **tasks_failed_count:** 0
- **tasks_completed_count:** 21

View workflow DAG -- colors grouped by apps
View workflow task summary

- Workflow resource usage: This visualization provides resource usage information at the workflow level. For eg, cumulative CPU/Memory utilization across workers over time.



test_fan_inout.py

- **Started:** 2019-06-13 10:58:14
- **Completed:** 2019-06-13 10:58:39
- **Workflow duration:** 0:00:25
- **Owner:** yadu
- **host:** borgmachine2
- **rundir:** /home/yadu/src/parsl/parsl/tests/manual_tests/
- **tasks_failed_count:** 0
- **tasks_completed_count:** 21

View workflow DAG -- colors grouped by apps
View workflow DAG -- colors grouped by task states
View workflow task summary

CPU Usage

Memory Usage

3.13 Container support

Containers provide an ideal way for abstracting execution resource heterogeneity and providing a common sandbox for execution.

There are two models for executing an app in a container:

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 approach can be used with other supported 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 how to create a pool of Docker containers, each with a worker that executes specific apps.

Installing Docker

To install Docker please ensure you have sudo privileges and follow Docker's installation 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, the following instructions might need to be tweaked for your specific system. Such cases will be noted explicitly.

1. Pull a Docker image with the latest Python.

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

2. Construct a new Python image 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 another Python version, make sure that the container built in the previous steps matches the host machine's environment.

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

6. Set up Parsl apps. The following directories contain sample apps for this guide:

   - `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 .

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

Parsl Config

Now that we have a Docker image available locally, we will create an `executor` 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.

```
from parsl.config import Config
from parsl.executors.ipp import IPyParallelExecutor
from libsubmit.providers.local.local import Local
```

(continues on next page)

(continued from previous page)

```
config = Config(
    executors=[
        IPyParallelExecutor(
            label='pool_app1',
            container_image='app1_v0.1'
            provider=Local(init_blocks=2)
        )
    ],
    lazy_errors=True
)
```

For workflows with multiple apps which require different Docker images, a new executor 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 `executors` keyword argument to ensure that apps execute on the specific executors 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--------+       |    +---------executor-1-------------+
|                   |       |    |           ...                  |
|                   |       |    | +-------App1Container--------+ |
| App1(executors=['pool1'])------+-+--------app1.py             | |
|                   |       |    | |         +-----predict()    | |
|       X           |       |    | +----------------------------+ |
|      / \          |       |    +--------------------------------+
|     Y...Y         |       |
|      \ /          |       |    +---------executor-2-------------+
|       Z           |       |    |           ...                  |
|                   |       |    | +-------App2Container--------+ |
| App2(executors=['pool2'])------+-+------- app2.py             | |
|                   |       |    | |         +-----predict()    | |
|                   |       |    | +----------------------------+ |
+-------------------+       |    +--------------------------------+
                            |
                            +------------------- -- -
```

The diagram above illustrates the various components and how they interact with each other to act as a fast model serving system. In this model, each executor in the Parsl config definition can only serve one container image. Parsl launches multiple blocks matching the definition of the executor, 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 is 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 executor that specifies the right container image app1_v0.1 is below :

```
@python_app(executors=['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())
```

3.14 Usage statistics collection

Parsl uses an **Opt-in** model to send anonymized usage statistics back to the Parsl development team to measure worldwide usage and improve reliability and usability. The usage statistics are used only for improvements and reporting. They are not shared in raw form outside of the Parsl team.

3.14.1 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

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 and is anonymized (see What is sent? below).

3.14.2 Opt-In

We have chosen opt-in collection rather than opt-out with the hope that workflow developers and researchers would choose to send us this information. The reason is that we need this data - it is a requirement for funding. We believe by leaving the decision to the users, we set a good balance between the benefits to the project and respecting the privacy of our users.

By opting-in, and allowing these statistics to be reported back, you are explicitly supporting the further development of Parsl.

If you wish to opt in to usage reporting, set `PARSL_TRACKING=true` in your environment or set `usage_tracking=True` in the configuration object (*parsl.config.Config*).

3.14.3 What is sent?

- Anonymized user ID
- Anonymized hostname
- Anonymized Parsl script ID
- Start and end times
- Parsl exit code
- Number of executors used
- Number of failures
- Parsl, libsubmit, Python version info
- OS and OS version

3.14.4 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.

3.14.5 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.

3.14.6 What will the data be used for?

The data will be used for reporting purposes to answer questions such as:

- How many unique users are using Parsl?
- To determine patterns of usage - is activity increasing or decreasing?

We will also use this information to improve Parsl by identifying software faults.

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

3.14.7 Feedback

Please send us your feedback at parsl@googlegroups.com. Feedback from our user communities will be useful in determining our path forward with usage tracking in the future. We do ask that if you have concerns or objections, please be specific in your feedback.

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.

```
import parsl

# 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.

```
@bash_app
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-executor support allows you to define the executor (including local threads) on which an App should be executed. For example:

```
@python_app(executors=['SuperComputer1'])
def BigSimulation(...):
    ...

@python_app(executors=['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 remote execution systems can identify the IP address automatically in certain cases, it is safer to specify the address explicitly. Parsl provides a few heuristic based address resolution methods that could be useful, however with complex networks some trial and error might be necessary to find the right address or network interface to use.

For *IPyParallelExecutor* the address is specified in the *Config* as shown below :

```
# THIS IS A CONFIG FRAGMENT FOR ILLUSTRATION
from parsl.config import Config
from parsl.executors import IPyParallelExecutor
from parsl.executors.ipp_controller import Controller
from parsl.addresses import address_by_route, address_by_query, address_by_hostname
config = Config(
    executors=[
        IPyParallelExecutor(
            label='ALCF_theta_local',
            controller=Controller(public_ip='<AA.BB.CC.DD>')          # specify␣
↪public ip here
            # controller=Controller(public_ip=address_by_route())      # Alternatively␣
↪you can try this
            # controller=Controller(public_ip=address_by_query())      # Alternatively␣
↪you can try this
            # controller=Controller(public_ip=address_by_hostname())   # Alternatively␣
↪you can try this
```

(continues on next page)

(continued from previous page)

```
        )
    ],
)


.. note::
   Another possibility that can cause workers not to connect back to Parsl is an␣
↪incompatibility between
   the system and the pre-compiled bindings used for pyzmq. As a last resort, you can␣
↪try:
   ``pip install --upgrade --no-binary pyzmq pyzmq``, which forces re-compilation.
```

For the *HighThroughputExecutor* as well as the *ExtremeScaleExecutor*, `address` is a keyword argument taken at initialization. Here is an example for the *HighThroughputExecutor*:

```
# THIS IS A CONFIG FRAGMENT FOR ILLUSTRATION
from parsl.config import Config
from parsl.executors import HighThroughputExecutor
from parsl.addresses import address_by_route, address_by_query, address_by_hostname

config = Config(
    executors=[
        HighThroughputExecutor(
            label='NERSC_Cori',
            address='<AA.BB.CC.DD>'            # specify public ip here
            # address=address_by_route()      # Alternatively you can try this
            # address=address_by_query()      # Alternatively you can try this
            # address=address_by_hostname()   # Alternatively you can try this
        )
    ],
)
```

Note: On certain systems such as the Midway RCC cluster at UChicago, some network interfaces have an active intrusion detection system that drops connections that persist beyond a specific duration (~20s). If you get repeated `ManagerLost` exceptions, it would warrant taking a closer look at networking.

4.7 parsl.dataflow.error.ConfigurationError

The Parsl configuration model underwent a major and non-backward compatible change in the transition to v0.6.0. Prior to v0.6.0 the configuration object was a python dictionary with nested dictionaries and lists. The switch to a class based configuration allowed for well-defined options for each specific component being configured as well as transparency on configuration defaults. The following traceback indicates that the old style configuration was passed to Parsl v0.6.0+ and requires an upgrade to the configuration.

```
File "/home/yadu/src/parsl/parsl/dataflow/dflow.py", line 70, in __init__
    'Expected `Config` class, received dictionary. For help, '
parsl.dataflow.error.ConfigurationError: Expected `Config` class, received dictionary.
↪ For help,
see http://parsl.readthedocs.io/en/stable/stubs/parsl.config.Config.html
```

For more information on how to update your configuration script, please refer to our configuration guide here.

4.8 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.

4.9 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 activate <my_env>

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

4.10 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'`

4.11 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 *parsl.dataflow.error.ConfigurationError*)

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

```
@bash_app
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)
```

4.12 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. See *Workers do not connect back to Parsl* for more details.
   - Firewall restrictions that block certain port ranges. If there is a certain port range that is **not** blocked, you may specify that via the `Controller` object:

   ```
   from libsubmit.providers import Cobalt
   from parsl.config import Config
   from parsl.executors.ipp import IPyParallelExecutor
   from parsl.executors.ipp_controller import Controller

   config = Config(
       executors=[
           IPyParallelExecutor(
               label='ALCF_theta_local',
               provider=Cobalt(),
               controller=Controller(port_range='50000,55000')
           )
       ],
   )
   ```

4.13 How can I start a Jupyter notebook over SSH?

Run

```
jupyter notebook --no-browser --ip=`/sbin/ip route get 8.8.8.8 | awk '{print $NF;exit}
↪'`
```

for a Jupyter notebook, or

```
jupyter lab --no-browser --ip=`/sbin/ip route get 8.8.8.8 | awk '{print $NF;exit}'`
```

for Jupyter lab (recommended). If that doesn't work, see instructions here.

4.14 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.

4.15 How do I cite Parsl?

To cite Parsl in publications, please use the following:

Babuji, Y., Woodard, A., Li, Z., Katz, D. S., Clifford, B., Kumar, R., Lacinski, L., Chard, R., Wozniak, J., Foster, I., Wilde, M., and Chard, K., Parsl: Pervasive Parallel Programming in Python. 28th ACM International Symposium on High-Performance Parallel and Distributed Computing (HPDC). 2019. https://doi.org/10.1145/3307681.3325400

or

```
@inproceedings{babuji19parsl,
  author        = {Babuji, Yadu and
                   Woodard, Anna and
                   Li, Zhuozhao and
                   Katz, Daniel S. and
                   Clifford, Ben and
                   Kumar, Rohan and
                   Lacinski, Lukasz and
                   Chard, Ryan and
                   Wozniak, Justin and
                   Foster, Ian and
                   Wilde, Mike and
                   Chard, Kyle},
  title         = {Parsl: Pervasive Parallel Programming in Python},
  booktitle     = {28th ACM International Symposium on High-Performance Parallel and
↪Distributed Computing (HPDC)},
  doi           = {10.1145/3307681.3325400},
  year          = {2019},
  url           = {https://doi.org/10.1145/3307681.3325400}
}
```

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.python_app`	Decorator function for making python apps.
`parsl.app.app.bash_app`	Decorator function for making bash apps.
`parsl.app.futures.DataFuture`	A datafuture points at an AppFuture.
`parsl.config.Config`	Specification of Parsl configuration options.
`parsl.dataflow.futures.AppFuture`	An AppFuture wraps a sequence of Futures which may fail and be retried.
`parsl.dataflow.dflow.DataFlowKernelLoader`	Manage which DataFlowKernel is active.
`parsl.data_provider.data_manager.DataManager`	The DataManager is responsible for transferring input and output data.
`parsl.data_provider.data_manager.Staging`	This class defines the interface for file staging providers.
`parsl.data_provider.files.File`	The Parsl File Class.
`parsl.executors.base.ParslExecutor`	Define the strict interface for all Executor classes.
`parsl.executors.ThreadPoolExecutor`	A thread-based executor.
`parsl.executors.IPyParallelExecutor`	The IPython Parallel executor.
`parsl.executors.ipp_controller.Controller`	Start and maintain a IPythonParallel controller.
`parsl.executors.HighThroughputExecutor`	Executor designed for cluster-scale
`parsl.executors.WorkQueueExecutor`	Executor to use Work Queue batch system
`parsl.executors.ExtremeScaleExecutor`	Executor designed for leadership class supercomputer scale
`parsl.executors.swift_t.TurbineExecutor`	The Turbine executor.
`parsl.channels.LocalChannel`	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

Continued on next page

Table 1 – continued from previous page

`parsl.channels.SSHChannel`	SSH persistent channel.
`parsl.channels.OAuthSSHChannel`	SSH persistent channel.
`parsl.channels.SSHInteractiveLoginChannel`	SSH persistent channel.
`parsl.providers.AdHocProvider`	Ad-hoc execution provider
`parsl.providers.AWSProvider`	A provider for using Amazon Elastic Compute Cloud (EC2) resources.
`parsl.providers.CobaltProvider`	Cobalt Execution Provider
`parsl.providers.CondorProvider`	HTCondor Execution Provider.
`parsl.providers.GoogleCloudProvider`	A provider for using resources from the Google Compute Engine.
`parsl.providers.GridEngineProvider`	A provider for the Grid Engine scheduler.
`parsl.providers.JetstreamProvider`	
`parsl.providers.LocalProvider`	Local Execution Provider
`parsl.providers.LSFProvider`	LSF Execution Provider
`parsl.providers.GridEngineProvider`	A provider for the Grid Engine scheduler.
`parsl.providers.SlurmProvider`	Slurm Execution Provider
`parsl.providers.TorqueProvider`	Torque Execution Provider
`parsl.providers.KubernetesProvider`	Kubernetes execution provider :param namespace: Kubernetes namespace to create deployments.
`parsl.providers.PBSProProvider`	PBS Pro Execution Provider
`parsl.launchers.SimpleLauncher`	Does no wrapping.
`parsl.launchers.SingleNodeLauncher`	Worker launcher that wraps the user's command with the framework to launch multiple command invocations in parallel.
`parsl.launchers.SrunLauncher`	Worker launcher that wraps the user's command with the SRUN launch framework to launch multiple cmd invocations in parallel on a single job allocation.
`parsl.launchers.AprunLauncher`	Worker launcher that wraps the user's command with the Aprun launch framework to launch multiple cmd invocations in parallel on a single job allocation
`parsl.launchers.SrunMPILauncher`	Launches as many workers as MPI tasks to be executed concurrently within a block.
`parsl.launchers.GnuParallelLauncher`	Worker launcher that wraps the user's command with the framework to launch multiple command invocations via GNU parallel sshlogin.
`parsl.launchers.MpiExecLauncher`	Worker launcher that wraps the user's command with the framework to launch multiple command invocations via mpiexec.
`parsl.launchers.JsrunLauncher`	Worker launcher that wraps the user's command with the Jsrun launch framework to launch multiple cmd invocations in parallel on a single job allocation
`parsl.monitoring.MonitoringHub`	

5.1 parsl.set_stream_logger

parsl.**set_stream_logger**(*name: str = 'parsl', level: int = 10, format_string: Optional[str] = None*)

Add a stream log handler.

Parameters

- **name** (–) – Set the logger name.
- **level** (–) – Set to logging.DEBUG by default.

- **format_string** (–) – Set to None by default.

Returns

- None

5.2 parsl.set_file_logger

parsl.**set_file_logger**(*filename: str*, *name: str = 'parsl'*, *level: int = 10*, *format_string: Optional[str] = None*)

Add a stream log handler.

Parameters

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

Returns

- None

5.3 parsl.app.app.python_app

parsl.app.app.**python_app**(*function=None*, *data_flow_kernel=None*, *walltime=60*, *cache=False*, *executors='all'*)

Decorator function for making python apps.

Parameters

- **function** (*function*) – Do not pass this keyword argument directly. This is needed in order to allow for omitted parenthesis, for example, `@python_app` if using all defaults or `@python_app(walltime=120)`. If the decorator is used alone, function will be the actual function being decorated, whereas if it is called with arguments, function will be None. Default is None.
- **data_flow_kernel** (`DataFlowKernel`) – The *DataFlowKernel* responsible for managing this app. This can be omitted only after calling `parsl.dataflow.dflow.DataFlowKernelLoader.load()`. Default is None.
- **walltime** (*int*) – Walltime for app in seconds. Default is 60.
- **executors** (*string or list*) – Labels of the executors that this app can execute over. Default is 'all'.
- **cache** (*bool*) – Enable caching of the app call. Default is False.

5.4 parsl.app.app.bash_app

parsl.app.app.**bash_app**(*function=None*, *data_flow_kernel=None*, *walltime=60*, *cache=False*, *executors='all'*)

Decorator function for making bash apps.

Parameters

- **function** (*function*) – Do not pass this keyword argument directly. This is needed in order to allow for omitted parenthesis, for example, `@bash_app` if using all defaults or `@bash_app(walltime=120)`. If the decorator is used alone, function will be the actual function being decorated, whereas if it is called with arguments, function will be None. Default is None.
- **data_flow_kernel** (`DataFlowKernel`) – The *DataFlowKernel* responsible for managing this app. This can be omitted only after calling `parsl.dataflow.dflow.DataFlowKernelLoader.load()`. Default is None.
- **walltime** (*int*) – Walltime for app in seconds. Default is 60.
- **executors** (*string or list*) – Labels of the executors that this app can execute over. Default is 'all'.
- **cache** (*bool*) – Enable caching of the app call. Default is False.

5.5 parsl.app.futures.DataFuture

class `parsl.app.futures.`**`DataFuture`**(*fut*, *file_obj*, *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*, *tid=None*)

Construct the DataFuture object.

If the file_obj is a string convert to a File.

Parameters

- **fut** (*–*) – AppFuture that this DataFuture will track
- **file_obj** (*–*) – Something representing file(s)

Kwargs:

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

Methods

`__init__`(fut, file_obj[, tid])	Construct the DataFuture object.
`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`()	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])	Return the result of the call that the future represents.
`running`()	Return True if the future is currently executing.

Continued on next page

Table 2 – continued from previous page

`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.

5.6 parsl.config.Config

class parsl.config.**Config**(*executors: Optional[List[parsl.executors.base.ParslExecutor]] = None*, *app_cache: bool = True*, *checkpoint_files: Optional[List[str]] = None*, *checkpoint_mode: Optional[str] = None*, *checkpoint_period: Optional[str] = None*, *data_management_max_threads: int = 10*, *lazy_errors: bool = True*, *retries: int = 0*, *run_dir: str = 'runinfo'*, *strategy: Optional[str] = 'simple'*, *max_idletime: float = 120.0*, *monitoring: Optional[parsl.monitoring.monitoring.MonitoringHub] = None*, *usage_tracking: bool = False*, *initialize_logging: bool = True*)

Specification of Parsl configuration options.

Parameters

- **executors** (*list of ParslExecutor, optional*) – List of executor instances to use. Possible executors include *ThreadPoolExecutor*, *IPyParallelExecutor*, or *TurbineExecutor*. Default is [*ThreadPoolExecutor()*].
- **app_cache** (*bool, optional*) – Enable app caching. Default is True.
- **checkpoint_files** (*list of str, optional*) – List of paths to checkpoint files. Default is None.
- **checkpoint_mode** (*str, optional*) – Checkpoint mode to use, can be 'dfk_exit', 'task_exit', or 'periodic'. If set to `None`, checkpointing will be disabled. Default is None.
- **checkpoint_period** (*str, optional*) – Time interval (in "HH:MM:SS") at which to checkpoint completed tasks. Only has an effect if `checkpoint_mode='periodic'`.
- **data_management_max_threads** (*int, optional*) – Maximum number of threads to allocate for the data manager to use for managing input and output transfers. Default is 10.
- **monitoring** (`MonitoringHub`, *optional*) – The config to use for database monitoring. Default is None which does not log to a database.
- **lazy_errors** (*bool, optional*) – If True, errors from task failures will not be raised until `future.result()` is called. Otherwise, they will be raised as soon as the task returns. Default is True.
- **retries** (*int, optional*) – Set the number of retries in case of failure. Default is 0.

- **run_dir** (*str, optional*) – Path to run directory. Default is 'runinfo'.
- **strategy** (*str, optional*) – Strategy to use for scaling resources according to workflow needs. Can be 'simple' or `None`. If `None`, dynamic scaling will be disabled. Default is 'simple'.
- **max_idletime** (*float, optional*) – The maximum idle time for an executor in the 'simple' strategy. Default is 120.0 seconds.
- **usage_tracking** (*bool, optional*) – Set this field to True to opt-in to Parsl's usage tracking system. Parsl only collects minimal, non personally-identifiable, information used for reporting to our funding agencies. Default is False.
- **initialize_logging** (*bool, optional*) – Make DFK optionally not initialize any logging. Log messages will still be passed into the python logging system under the `parsl` logger name, but the logging system will not by default perform any further log system configuration. Most noticeably, it will not create a parsl.log logfile. The use case for this is when parsl is used as a library in a bigger system which wants to configure logging in a way that makes sense for that bigger system as a whole.

__init__(*executors: Optional[List[parsl.executors.base.ParslExecutor]] = None*, *app_cache: bool = True*, *checkpoint_files: Optional[List[str]] = None*, *checkpoint_mode: Optional[str] = None*, *checkpoint_period: Optional[str] = None*, *data_management_max_threads: int = 10*, *lazy_errors: bool = True*, *retries: int = 0*, *run_dir: str = 'runinfo'*, *strategy: Optional[str] = 'simple'*, *max_idletime: float = 120.0*, *monitoring: Optional[parsl.monitoring.monitoring.MonitoringHub] = None*, *usage_tracking: bool = False*, *initialize_logging: bool = True*)

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

Methods

`__init__`(executors, app_cache, ...)	Initialize self.

Attributes

`executors`

5.7 parsl.dataflow.futures.AppFuture

class `parsl.dataflow.futures.`**AppFuture**(*task_def*)

An AppFuture wraps a sequence of Futures which may fail and be retried.

The AppFuture will wait for the DFK to provide a result from an appropriate parent future, through `parent_callback`. It will set its result to the result of that parent future, if that parent future completes without an exception. This result setting should cause .result(), .exception() and done callbacks to fire as expected.

The AppFuture will not set its result to the result of the parent future, if that parent future completes with an exception, and if that parent future has retries left. In that case, no result(), exception() or done callbacks should report a result.

The AppFuture will set its result to the result of the parent future, if that parent future completes with an exception and if that parent future has no retries left, or if it has no retry field. .result(), .exception() and done callbacks should give a result as expected when a Future has a result set

The parent future may return a RemoteExceptionWrapper as a result and AppFuture will treat this an an exception for the above retry and result handling behaviour.

`__init__`(*task_def*)
Initialize the AppFuture.

Args:

KWargs:

- **task_def** [The DFK task definition dictionary for the task represented] by this future.

Methods

`__init__`(task_def)	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`()	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`(executor_fu)	Callback from a parent future to update the AppFuture.
`result`([timeout])	Return the result of the call that the future represents.
`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.
`AppFuture.update_parent`	

Attributes

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

5.8 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

`clear()`	Clear the active DataFlowKernel so that a new one can be loaded.
`dfk()`	Return the currently-loaded DataFlowKernel.
`load(config)`	Load a DataFlowKernel.
`wait_for_current_tasks()`	Waits for all tasks in the task list to be completed, by waiting for their AppFuture to be completed.

5.9 parsl.data_provider.data_manager.DataManager

class `parsl.data_provider.data_manager.`**`DataManager`**(*dfk: DataFlowKernel*)

The DataManager is responsible for transferring input and output data.

`__init__`(*dfk: DataFlowKernel*) → None

Initialize the DataManager.

Parameters **`dfk`** (*–*) – The DataFlowKernel that this DataManager is managing data for.

Methods

`__init__`(dfk)	Initialize the DataManager.
`DataManager.add_file`	
`DataManager.get_data_manager`	
`DataManager.scale_in`	
`DataManager.scale_out`	
`DataManager.shutdown`	
`stage_in(file, input, executor)`	Transport the input from the input source to the executor, if it is file-like, returning a DataFuture that wraps the stage-in operation.
`stage_out(file, executor, app_fu)`	Transport the file from the local filesystem to the remote Globus endpoint.
`DataManager.start`	
`DataManager.submit`	

Attributes

`DataManager.run_dir`
`DataManager.scaling_enabled`

5.10 parsl.data_provider.data_manager.Staging

class `parsl.data_provider.data_manager.`**`Staging`**

This class defines the interface for file staging providers.

For each file to be staged in, the data manager will present the file to each configured Staging provider in turn: first, it will ask if the provider can stage this file by calling `can_stage_in`, and if so, it will call both `stage_in` and `replace_task` to give the provider the opportunity to perform staging.

For each file to be staged out, the data manager will follow the same pattern using the corresponding stage out methods of this class.

The default implementation of this class rejects all files, and performs no staging actions.

To implement a concrete provider, one or both of the `can_stage_*` methods should be overridden to match the appropriate files, and then the corresponding `stage_*` and/or `replace_task*` methods should be implemented.

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

Methods

`can_stage_in`(file)	Given a File object, decide if this staging provider can stage the file.
`can_stage_out`(file)	Like can_stage_in, but for staging out.
`replace_task`(dm, executor, file, func)	For a file to be staged in, optionally return a replacement app function, which usually should be the original app function wrapped in staging code.
`replace_task_stage_out`(dm, executor, file, func)	For a file to be staged out, optionally return a replacement app function, which usually should be the original app function wrapped in staging code.
`stage_in`(dm, executor, file, parent_fut)	This call gives the staging provider an opportunity to prepare for stage-in and to launch arbitrary tasks which must complete as part of stage-in.
`stage_out`(dm, executor, file, app_fu)	This call gives the staging provider an opportunity to prepare for stage-out and to launch arbitrary tasks which must complete as part of stage-out.

5.11 parsl.data_provider.files.File

class `parsl.data_provider.files.`**`File`**`(`*url: str*`)`
The Parsl File Class.

This represents the global, and sometimes local, URI/path to a file.

Staging-in mechanisms may annotate a file with a local path recording the path at the far end of a staging action. It is up to the user of the File object to track which local scope that local path actually refers to.

`__init__``(`*url: str*`)`
Construct a File object from a url string.

Parameters **`url`** (–) – 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'

Methods

`__init__`(url)	Construct a File object from a url string.
`File.capitalize`	
`File.casefold`	
`File.center`	

Continued on next page

Table 12 – continued from previous page

`File.count`
`File.encode`
`File.endswith`
`File.expandtabs`
`File.find`
`File.format`
`File.format_map`
`File.get_data_future`
`File.index`
`File.is_remote`
`File.isalnum`
`File.isalpha`
`File.isdecimal`
`File.isdigit`
`File.isidentifier`
`File.islower`
`File.isnumeric`
`File.isprintable`
`File.isspace`
`File.istitle`
`File.isupper`
`File.join`
`File.ljust`
`File.lower`
`File.lstrip`
`File.maketrans`
`File.partition`
`File.replace`
`File.rfind`
`File.rindex`
`File.rjust`
`File.rpartition`
`File.rsplit`
`File.rstrip`
`File.set_data_future`
`File.split`
`File.splitlines`
`File.stage_in`
`File.stage_out`
`File.startswith`
`File.strip`
`File.swapcase`
`File.title`
`File.translate`
`File.upper`
`File.zfill`

Attributes

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

5.12 parsl.executors.base.ParslExecutor

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

Define the strict interface for all Executor classes.

This is a metaclass that only enforces concrete implementations of functionality by the child classes.

In addition to the listed methods, a ParslExecutor instance must always have a member field:

label: str - a human readable label for the executor, unique with respect to other executors.

An executor may optionally expose:

storage_access: List[parsl.data_provider.staging.Staging] - a list of staging providers that will be used for file staging. In the absence of this attribute, or if this attribute is `None`, then a default value of `parsl.data_provider.staging.default_staging` will be used by the staging code.

Typechecker note: Ideally storage_access would be declared on executor __init__ methods as List[Staging] - however, lists are by default invariant, not co-variant, and it looks like @typeguard cannot be persuaded otherwise. So if you're implementing an executor and want to @typeguard the constructor, you'll have to use List[Any] here.

`__init__``()`

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

Methods

`scale_in`(blocks)	Scale in method.
`scale_out`(blocks)	Scale out method.
`shutdown`()	Shutdown the executor.
`start`()	Start the executor.
`submit`(func, *args, **kwargs)	Submit.

Attributes

`run_dir`	Path to the run directory.
`scaling_enabled`	Specify if scaling is enabled.

5.13 parsl.executors.ThreadPoolExecutor

class `parsl.executors.`**`ThreadPoolExecutor`**(*label: str = 'threads', max_threads: int = 2, thread_name_prefix: str = '', storage_access: List[Any] = None, working_dir: Optional[str] = None, managed: bool = True*)

A thread-based executor.

Parameters

- **`max_threads`** (*`int`*) – Number of threads. Default is 2.

- **thread_name_prefix** (*string*) – Thread name prefix (only supported in python v3.6+).
- **storage_access** (list of `Staging`) – Specifications for accessing data this executor remotely.
- **managed** (*bool*) – If True, parsl will control dynamic scaling of this executor, and be responsible. Otherwise, this is managed by the user.

__init__(*label: str = 'threads', max_threads: int = 2, thread_name_prefix: str = '', storage_access: List[Any] = None, working_dir: Optional[str] = None, managed: bool = True*)
Initialize self. See help(type(self)) for accurate signature.

Methods

`__init__`(label, max_threads, ...)	Initialize self.
`scale_in`(blocks)	Scale in the number of active blocks by specified amount.
`scale_out`([workers])	Scales out the number of active workers by 1.
`shutdown`([block])	Shutdown the ThreadPool.
`start`()	Start the executor.
`submit`(*args, **kwargs)	Submits work to the thread pool.

Attributes

`run_dir`	Path to the run directory.
`scaling_enabled`	Specify if scaling is enabled.

5.14 parsl.executors.IPyParallelExecutor

class parsl.executors.**IPyParallelExecutor**(*provider=LocalProvider(channel=LocalChannel(envs={}, script_dir=None, userhome='/home/docs/checkouts/readthedocs.org/user_builds/parsl/checko), cmd_timeout=30, init_blocks=4, launcher=SingleNodeLauncher(), max_blocks=10, min_blocks=0, move_files=None, nodes_per_block=1, parallelism=1, walltime='00:15:00', worker_init=''), label='ipp', working_dir=None, controller=Controller(interfaces=None, ipython_dir='~/.ipython', log=True, mode='auto', port=None, port_range=None, profile='default', public_ip=None, reuse=False), container_image=None, engine_dir=None, storage_access=None, engine_debug_level=None, workers_per_node=1, managed=True*)

The IPython Parallel executor.

This executor uses IPythonParallel's pilot execution system to manage multiple processes running locally or remotely.

Parameters

- **provider** (*ExecutionProvider*) – Provider to access computation resources. Can be one of `EC2Provider`, `Cobalt`, `Condor`, `GoogleCloud`, `GridEngine`, `Jetstream`, `Local`, `GridEngine`, `Slurm`, or `Torque`.
- **label** (*str*) – Label for this executor instance.
- **controller** (*Controller*) – Which Controller instance to use. Default is `Controller()`.
- **workers_per_node** (*int*) – Number of workers to be launched per node. Default=1
- **container_image** (*str*) – Launch tasks in a container using this docker image. If set to None, no container is used. Default is None.
- **engine_dir** (*str*) – Directory where engine logs and configuration files will be stored.
- **working_dir** (*str*) – Directory where input data should be staged to.
- **storage_access** (list of `Staging`) – Specifications for accessing data this executor remotely.
- **managed** (*bool*) – If True, parsl will control dynamic scaling of this executor, and be responsible. Otherwise, this is managed by the user.
- **engine_debug_level** (*int | str*) – Sets engine logging to specified debug level. Choices: (0, 10, 20, 30, 40, 50, 'DEBUG', 'INFO', 'WARN', 'ERROR', 'CRITICAL')

:param .. note::: Some deficiencies with this executor are:

1. Ipengines 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__(*provider=LocalProvider(channel=LocalChannel(envs={}, script_dir=None, userhome='/home/docs/checkouts/readthedocs.org/user_builds/parsl/checkouts/0.9.0/docs'), cmd_timeout=30, init_blocks=4, launcher=SingleNodeLauncher(), max_blocks=10, min_blocks=0, move_files=None, nodes_per_block=1, parallelism=1, walltime='00:15:00', worker_init=''), label='ipp', working_dir=None, controller=Controller(interfaces=None, ipython_dir='~/.ipython', log=True, mode='auto', port=None, port_range=None, profile='default', public_ip=None, reuse=False), container_image=None, engine_dir=None, storage_access=None, engine_debug_level=None, workers_per_node=1, managed=True*)

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

Methods

`__init__`([provider, label, working_dir, . . .])	Initialize self.
`compose_containerized_launch_cmd`(filepath, . . .)	Reads the json contents from filepath and uses that to compose the engine launch command.
`compose_launch_cmd`(filepath, engine_dir, . . .)	Reads the json contents from filepath and uses that to compose the engine launch command.
`scale_in`(blocks)	Scale in the number of active blocks by the specified number.

Continued on next page

Table 18 – continued from previous page

scale_out([blocks])	Scales out the number of active workers by 1.
shutdown([hub, targets, block])	Shutdown the executor, including all workers and controllers.
start()	Start the executor.
status()	Returns the status of the executor via probing the execution providers.
submit(*args, **kwargs)	Submits work to the thread pool.

Attributes

connected_workers	
outstanding	
run_dir	Path to the run directory.
scaling_enabled	Specify if scaling is enabled.

5.15 parsl.executors.ipp_controller.Controller

class parsl.executors.ipp_controller.**Controller**(*public_ip=None*, *interfaces=None*, *port=None*, *port_range=None*, *reuse=False*, *log=True*, *ipython_dir='~/.ipython'*, *mode='auto'*, *profile='default'*)

Start and maintain a IPythonParallel controller.

Parameters

- **public_ip** (*str, optional*) – Specific IP address of the controller, as seen from the engines. If None, an attempt will be made to guess the correct value. Default is None.
- **interfaces** (*str, optional*) – Interfaces for ZeroMQ to listen on. Default is "*".
- **port** (*int or str, optional*) – Port on which the iPython hub listens for registration. If set to None, the IPython default will be used. Default is None.
- **port_range** (*str, optional*) – The minimum and maximum port values to use, in the format '<min>,<max>' (for example: '50000,60000'). If this does not equal None, random ports in port_range will be selected for all HubFactory listening services. This option overrides the port setting value for registration.
- **reuse** (*bool, optional*) – Reuse an existing controller.
- **ipython_dir** (*str, optional*) – IPython directory for IPythonParallel to store config files. This will be overriden by the auto controller start. Default is "~/.ipython".
- **profile** (*str, optional*) – Path to an IPython profile to use. Default is 'default'.
- **mode** (*str, optional*) – If "auto", controller will be created and managed automatically. If "manual" the controller is assumed to be created by the user. Default is auto.

__init__(*public_ip=None*, *interfaces=None*, *port=None*, *port_range=None*, *reuse=False*, *log=True*, *ipython_dir='~/.ipython'*, *mode='auto'*, *profile='default'*)

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

Methods

`__init__`([public_ip, interfaces, port, ...])	Initialize self.
`close()`	Terminate the controller process and its child processes.
`start()`	Start the controller.

Attributes

`client_file`	Specify path to the ipcontroller-client.json file.
`engine_file`	Specify path to the ipcontroller-engine.json file.

5.16 parsl.executors.HighThroughputExecutor

class `parsl.executors.`**`HighThroughputExecutor`**(*label: str = 'HighThroughputExecutor', provider: parsl.providers.provider_base.ExecutionProvider = LocalProvider(channel=LocalChannel(envs={}, script_dir=None, userhome='/home/docs/checkouts/readthedocs.org/user_builds/parsl/ch), cmd_timeout=30, init_blocks=4, launcher=SingleNodeLauncher(), max_blocks=10, min_blocks=0, move_files=None, nodes_per_block=1, parallelism=1, walltime='00:15:00', worker_init=''), launch_cmd: Optional[str] = None, address: str = '127.0.0.1', worker_ports: Optional[Tuple[int, int]] = None, worker_port_range: Optional[Tuple[int, int]] = (54000, 55000), interchange_port_range: Optional[Tuple[int, int]] = (55000, 56000), storage_access: Optional[List[parsl.data_provider.staging.Staging]] = None, working_dir: Optional[str] = None, worker_debug: bool = False, cores_per_worker: float = 1.0, mem_per_worker: Optional[float] = None, max_workers: Union[int, float] = inf, prefetch_capacity: int = 0, heartbeat_threshold: int = 120, heartbeat_period: int = 30, poll_period: int = 10, suppress_failure: bool = True, managed: bool = True, worker_logdir_root: Optional[str] = None*)

Executor designed for cluster-scale

The HighThroughputExecutor system has the following components:

1. The HighThroughputExecutor instance which is run as part of the Parsl script.

2. The Interchange which is acts as a load-balancing proxy between workers and Parsl
3. The multiprocessing based worker pool which coordinates task execution over several cores on a node.
4. ZeroMQ pipes connect the HighThroughputExecutor, Interchange and the process_worker_pool

Here is a diagram

```
             |  Data   |  Executor   |  Interchange  | External Process(es)
             |  Flow   |             |               |
        Task | Kernel  |             |               |
      +----->|-------->|------------>|->outgoing_q---|-> process_worker_pool
      |      |         |             | batching      |    |         |
Parsl<---Fut-|         |             | load-balancing|  result   exception
          ^  |         |             | watchdogs     |    |         |
          |  |         |   Q_mngmnt  |               |    V         V
          |  |         |    Thread<--|-incoming_q<---|--- +---------+
          |  |         |      |      |               |
          |  |         |      |      |               |
          +----update_fut-----+
```

Each of the workers in each process_worker_pool has access to its local rank through an environmental variable, `PARSL_WORKER_RANK`. The local rank is unique for each process and is an integer in the range from 0 to the number of workers per in the pool minus 1. The workers also have access to the ID of the worker pool as `PARSL_WORKER_POOL_ID` and the size of the worker pool as `PARSL_WORKER_COUNT`.

Parameters

- **provider** (*ExecutionProvider*) –

  Provider to access computation resources. Can be one of `EC2Provider`, `Cobalt`, `Condor`, `GoogleCloud`, `GridEngine`, `Jetstream`, `Local`, `GridEngine`, `Slurm`, or `Torque`.
- **label** (*str*) – Label for this executor instance.
- **launch_cmd** (*str*) – Command line string to launch the process_worker_pool from the provider. The command line string will be formatted with appropriate values for the following values (debug, task_url, result_url, cores_per_worker, nodes_per_block, heartbeat_period ,heartbeat_threshold, logdir). For example: launch_cmd="process_worker_pool.py {debug} -c {cores_per_worker} –task_url={task_url} –result_url={result_url}"
- **address** (*string*) – An address to connect to the main Parsl process which is reachable from the network in which workers will be running. This can be either a hostname as returned by `hostname` or an IP address. Most login nodes on clusters have several network interfaces available, only some of which can be reached from the compute nodes. Some trial and error might be necessary to identify what addresses are reachable from compute nodes.
- **worker_ports** (*(int, int)*) – Specify the ports to be used by workers to connect to Parsl. If this option is specified, worker_port_range will not be honored.
- **worker_port_range** (*(int, int)*) – Worker ports will be chosen between the two integers provided.
- **interchange_port_range** (*(int, int)*) – Port range used by Parsl to communicate with the Interchange.
- **working_dir** (*str*) – Working dir to be used by the executor.
- **worker_debug** (*Bool*) – Enables worker debug logging.
- **managed** (*Bool*) – If this executor is managed by the DFK or externally handled.

- **cores_per_worker** (*float*) – cores to be assigned to each worker. Oversubscription is possible by setting cores_per_worker < 1.0. Default=1
- **mem_per_worker** (*float*) – GB of memory required per worker. If this option is specified, the node manager will check the available memory at startup and limit the number of workers such that the there's sufficient memory for each worker. Default: None
- **max_workers** (*int*) – Caps the number of workers launched by the manager. Default: infinity
- **prefetch_capacity** (*int*) – Number of tasks that could be prefetched over available worker capacity. When there are a few tasks (<100) or when tasks are long running, this option should be set to 0 for better load balancing. Default is 0.
- **suppress_failure** (*Bool*) – If set, the interchange will suppress failures rather than terminate early. Default: True
- **heartbeat_threshold** (*int*) – Seconds since the last message from the counterpart in the communication pair: (interchange, manager) after which the counterpart is assumed to be un-available. Default: 120s
- **heartbeat_period** (*int*) – Number of seconds after which a heartbeat message indicating liveness is sent to the counterpart (interchange, manager). Default: 30s
- **poll_period** (*int*) – Timeout period to be used by the executor components in milliseconds. Increasing poll_periods trades performance for cpu efficiency. Default: 10ms
- **worker_logdir_root** (*string*) – In case of a remote file system, specify the path to where logs will be kept.

__init__(*label: str = 'HighThroughputExecutor', provider: parsl.providers.provider_base.ExecutionProvider = LocalProvider(channel=LocalChannel(envs={}, script_dir=None, userhome='/home/docs/checkouts/readthedocs.org/user_builds/parsl/checkouts/0.9.0/docs'), cmd_timeout=30, init_blocks=4, launcher=SingleNodeLauncher(), max_blocks=10, min_blocks=0, move_files=None, nodes_per_block=1, parallelism=1, walltime='00:15:00', worker_init=''), launch_cmd: Optional[str] = None, address: str = '127.0.0.1', worker_ports: Optional[Tuple[int, int]] = None, worker_port_range: Optional[Tuple[int, int]] = (54000, 55000), interchange_port_range: Optional[Tuple[int, int]] = (55000, 56000), storage_access: Optional[List[parsl.data_provider.staging.Staging]] = None, working_dir: Optional[str] = None, worker_debug: bool = False, cores_per_worker: float = 1.0, mem_per_worker: Optional[float] = None, max_workers: Union[int, float] = inf, prefetch_capacity: int = 0, heartbeat_threshold: int = 120, heartbeat_period: int = 30, poll_period: int = 10, suppress_failure: bool = True, managed: bool = True, worker_logdir_root: Optional[str] = None*)

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

Methods

`__init__`(label, provider, script_dir=None, ...)	Initialize self.
`hold_worker`(worker_id)	Puts a worker on hold, preventing scheduling of additional tasks to it.
`initialize_scaling`()	Compose the launch command and call the scale_out
`scale_in`([blocks, block_ids])	Scale in the number of active blocks by specified amount.
`scale_out`([blocks])	Scales out the number of blocks by "blocks"

Continued on next page

Table 22 – continued from previous page

`shutdown`([hub, targets, block])	Shutdown the executor, including all workers and controllers.
`start`()	Create the Interchange process and connect to it.
`status`()	Return status of all blocks.
`submit`(func, *args, **kwargs)	Submits work to the the outgoing_q.
`weakref_cb`([q])	We do not use this yet.

Attributes

`connected_workers`	
`outstanding`	
`run_dir`	Path to the run directory.
`scaling_enabled`	Specify if scaling is enabled.

5.17 parsl.executors.WorkQueueExecutor

class parsl.executors.**WorkQueueExecutor**(*label='WorkQueueExecutor'*, *working_dir='.'*, *managed=True*, *project_name=None*, *project_password=None*, *project_password_file=None*, *port=0*, *env=None*, *shared_fs=False*, *source=False*, *init_command=''*, *full_debug=True*, *see_worker_output=False*)

Executor to use Work Queue batch system

The WorkQueueExecutor system utilizes the Work Queue framework to efficiently delegate Parsl apps to remote machines in clusters and grids using a fault-tolerant system. Users can run the work_queue_worker program on remote machines to connect to the WorkQueueExecutor, and Parsl apps will then be sent out to these machines for execution and retrieval.

label: str A human readable label for the executor, unique with respect to other Work Queue master programs

working_dir: str Location for Parsl to perform app delegation to the Work Queue system

managed: bool If this executor is managed by the DFK or externally handled

project_name: str Work Queue process name

project_password: str Optional password for the Work Queue project

project_password_file: str Optional password file for the work queue project

port: int TCP port on Parsl submission machine for Work Queue workers to connect to. Workers will specify this port number when trying to connect to Parsl

env: dict{str} Dictionary that contains the environmental variables that need to be set on the Work Queue worker machine

shared_fs: bool Define if working in a shared file system or not. If Parsl and the Work Queue workers are on a shared file system, Work Queue does not need to transfer and rename files for execution

source: bool Choose whether to transfer parsl app information as source code. (Note: this increases throughput for @python_apps, but the implementation does not include functionality for @bash_apps, and thus source=False must be used for programs utilizing @bash_apps.)

init_command: str Command to run before constructed Work Queue command

see_worker_output: bool Prints worker standard output if true

`__init__`(*label='WorkQueueExecutor', working_dir='.', managed=True, project_name=None, project_password=None, project_password_file=None, port=0, env=None, shared_fs=False, source=False, init_command='', full_debug=True, see_worker_output=False*)

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

Methods

`__init__`([label, working_dir, managed, ...])	Initialize self.
`create_name_tuple`(parsl_file_obj, in_or_out)	Returns a tuple containing information about an input or output file to a Parsl app.
`create_new_name`(file_name)	Returns a unique file name for an input file name.
`run_dir`([value])	Path to the run directory.
`scale_in`(count)	Scale in method.
`scale_out`(*args, **kwargs)	Scale out method.
`scaling_enabled`()	Specify if scaling is enabled.
`shutdown`(*args, **kwargs)	Shutdown the executor.
`start`()	Create submit process and collector thread to create, send, and retrieve Parsl tasks within the Work Queue system.
`submit`(func, *args, **kwargs)	Processes the Parsl app by its arguments and submits the function information to the task queue, to be executed using the Work Queue system.

Attributes

`hub_address`	Address to the Hub for monitoring.
`hub_port`	Port to the Hub for monitoring.

5.18 parsl.executors.ExtremeScaleExecutor

class parsl.executors.**ExtremeScaleExecutor**(*label='ExtremeScaleExecutor', provider=LocalProvider(channel=LocalChannel(envs={}, script_dir=None, userhome='/home/docs/checkouts/readthedocs.org/user_builds/parsl/check), cmd_timeout=30, init_blocks=4, launcher=SingleNodeLauncher(), max_blocks=10, min_blocks=0, move_files=None, nodes_per_block=1, parallelism=1, walltime='00:15:00', worker_init=''), launch_cmd=None, address='127.0.0.1', worker_ports=None, worker_port_range=(54000, 55000), interchange_port_range=(55000, 56000), storage_access=None, working_dir=None, worker_debug=False, ranks_per_node=1, heartbeat_threshold=120, heartbeat_period=30, managed=True*)

Executor designed for leadership class supercomputer scale

The ExtremeScaleExecutor extends the Executor interface to enable task execution on supercomputing systems (>1K Nodes). When functions and their arguments are submitted to the interface, a future is returned that tracks the execution of the function on a distributed compute environment.

The ExtremeScaleExecutor system has the following components:

1. The ExtremeScaleExecutor instance which is run as part of the Parsl script
2. The Interchange which is acts as a load-balancing proxy between workers and Parsl
3. The MPI based mpi_worker_pool which coordinates task execution over several nodes With MPI communication between workers, we can exploit low latency networking on HPC systems.
4. ZeroMQ pipes that connect the ExtremeScaleExecutor, Interchange and the mpi_worker_pool

Our design assumes that there is a single MPI application (mpi_worker_pool) running over a `block` and that there might be several such instances.

Here is a diagram

```
             |  Data   |  Executor   |  Interchange  | External Process(es)
             |  Flow   |             |               |
        Task | Kernel  |             |               |
      +----->|-------->|------------>|->outgoing_q---|-> mpi_worker_pool
      |      |         |             | batching      |    |         |
Parsl<---Fut-|         |             | load-balancing|  result   exception
          ^  |         |             | watchdogs     |    |         |
          |  |         |   Q_mngmnt  |               |    V         V
          |  |         |    Thread<--|-incoming_q<---|--- +---------+
          |  |         |      |      |               |
          |  |         |      |      |               |
          +----update_fut-----+
```

Parameters

- **provider** (*ExecutionProvider*) –

Provider to access computation resources. Can be any providers in `parsl.providers`: `Cobalt`, `Condor`, `GoogleCloud`, `GridEngine`, `Jetstream`, `Local`, `GridEngine`, `Slurm`, or `Torque`.

- **`label`** (*`str`*) – Label for this executor instance.
- **`launch_cmd`** (*`str`*) – Command line string to launch the mpi_worker_pool from the provider. The command line string will be formatted with appropriate values for the following values (debug, task_url, result_url, ranks_per_node, nodes_per_block, heartbeat_period ,heartbeat_threshold, logdir). For example: launch_cmd="mpiexec -np {ranks_per_node} mpi_worker_pool.py {debug} –task_url={task_url} –result_url={result_url}"
- **`address`** (*`string`*) – An address to connect to the main Parsl process which is reachable from the network in which workers will be running. This can be either a hostname as returned by `hostname` or an IP address. Most login nodes on clusters have several network interfaces available, only some of which can be reached from the compute nodes. Some trial and error might be necessary to identify what addresses are reachable from compute nodes.
- **`worker_ports`** (*`(int, int)`*) – Specify the ports to be used by workers to connect to Parsl. If this option is specified, worker_port_range will not be honored.
- **`worker_port_range`** (*`(int, int)`*) – Worker ports will be chosen between the two integers provided.
- **`interchange_port_range`** (*`(int, int)`*) – Port range used by Parsl to communicate with the Interchange.
- **`working_dir`** (*`str`*) – Working dir to be used by the executor.
- **`worker_debug`** (*`Bool`*) – Enables engine debug logging.
- **`managed`** (*`Bool`*) – If this executor is managed by the DFK or externally handled.
- **`ranks_per_node`** (*`int`*) – Specify the ranks to be launched per node.
- **`heartbeat_threshold`** (*`int`*) – Seconds since the last message from the counterpart in the communication pair: (interchange, manager) after which the counterpart is assumed to be un-available. Default:120s
- **`heartbeat_period`** (*`int`*) – Number of seconds after which a heartbeat message indicating liveness is sent to the counterpart (interchange, manager). Default:30s

`__init__`(*label='ExtremeScaleExecutor', provider=LocalProvider(channel=LocalChannel(envs={}, script_dir=None, userhome='/home/docs/checkouts/readthedocs.org/user_builds/parsl/checkouts/0.9.0/docs'), cmd_timeout=30, init_blocks=4, launcher=SingleNodeLauncher(), max_blocks=10, min_blocks=0, move_files=None, nodes_per_block=1, parallelism=1, walltime='00:15:00', worker_init=''), launch_cmd=None, address='127.0.0.1', worker_ports=None, worker_port_range=(54000, 55000), interchange_port_range=(55000, 56000), storage_access=None, working_dir=None, worker_debug=False, ranks_per_node=1, heartbeat_threshold=120, heartbeat_period=30, managed=True*)

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

Methods

`__init__`([label, provider, launch_cmd, …])	Initialize self.

Continued on next page

Table 26 – continued from previous page

hold_worker(worker_id)	Puts a worker on hold, preventing scheduling of additional tasks to it.
initialize_scaling()	Compose the launch command and call the scale_out
scale_in([blocks, block_ids])	Scale in the number of active blocks by specified amount.
scale_out([blocks])	Scales out the number of blocks by "blocks"
shutdown([hub, targets, block])	Shutdown the executor, including all workers and controllers.
start()	Create the Interchange process and connect to it.
status()	Return status of all blocks.
submit(func, *args, **kwargs)	Submits work to the the outgoing_q.
weakref_cb([q])	We do not use this yet.

Attributes

connected_workers	
outstanding	
run_dir	Path to the run directory.
scaling_enabled	Specify if scaling is enabled.

5.19 parsl.executors.swift_t.TurbineExecutor

class parsl.executors.swift_t.**TurbineExecutor**(*label='turbine'*, *storage_access=None*, *working_dir=None*, *managed=True*)

The Turbine executor.

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

Here is a diagram

```
             |  Data   | Executor    |   IPC      | External Process(es)
             |  Flow   |             |            |
        Task | Kernel  |             |            |
      +----->|-------->|------------>|outgoing_q -|-> Worker_Process
      |      |         |             |            |    |         |
Parsl<---Fut-|         |             |            |  result   exception
          ^  |         |             |            |    |         |
          |  |         |   Q_mngmnt  |            |    V         V
          |  |         |    Thread<--|incoming_q<-|--- +---------+
          |  |         |      |      |            |
          |  |         |      |      |            |
          +----update_fut-----+
```

__init__(*label='turbine'*, *storage_access=None*, *working_dir=None*, *managed=True*)

Initialize the thread pool.

Trying to implement the emews model.

Methods

`__init__`([label, storage_access, . . .])	Initialize the thread pool.
`scale_in`(blocks)	Scale in the number of active blocks by specified amount.
`scale_out`([blocks])	Scales out the number of active workers by 1.
`shutdown`()	Shutdown method, to kill the threads and workers.
`start`()	Start the executor.
`submit`(func, *args, **kwargs)	Submits work to the the outgoing_q.
`weakref_cb`([q])	We do not use this yet.

Attributes

`run_dir`	Path to the run directory.
`scaling_enabled`	Specify if scaling is enabled.

5.20 parsl.channels.LocalChannel

class `parsl.channels.`**`LocalChannel`**(*userhome='.', envs={}, script_dir=None, **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={}, script_dir=None, **kwargs*)

Initialize the local channel. script_dir 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
- script_dir (string): Directory to place scripts

Methods

`__init__`([userhome, envs, script_dir])	Initialize the local channel.
`abspath`(path)	Return the absolute path.
`close`()	There's nothing to close here, and this really doesn't do anything
`execute_no_wait`(cmd, walltime[, envs])	Synchronously execute a commandline string on the shell.
`execute_wait`(cmd[, walltime, envs])	Synchronously execute a commandline string on the shell.
`isdir`(path)	Return true if the path refers to an existing directory.
`makedirs`(path[, mode, exist_ok])	Create a directory.
`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.

Attributes

`script_dir`	This is a property.

5.21 parsl.channels.SSHChannel

class parsl.channels.**SSHChannel**(*hostname*, *username=None*, *password=None*, *script_dir=None*, *envs=None*, *gssapi_auth=False*, *skip_auth=False*, *port=22*, ***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*, *script_dir=None*, *envs=None*, *gssapi_auth=False*, *skip_auth=False*, *port=22*, ***kwargs*)

Initialize a persistent connection to the remote system. We should know at this point whether ssh connectivity is possible

Parameters **hostname** (–) – Hostname

KWargs:

- username (string) : Username on remote system
- password (string) : Password for remote system
- port : The port designated for the ssh connection. Default is 22.
- script_dir (string) : Full path to a script dir where generated scripts could be sent to.
- envs (dict) : A dictionary of environment variables to be set when executing commands

Raises:

Methods

__init__(hostname[, username, password, ...])	Initialize a persistent connection to the remote system.
abspath(path)	Return the absolute path on the remote side.
close()	Closes the channel.
execute_no_wait(cmd[, walltime, envs])	Execute asynchronousely without waiting for exitcode
execute_wait(cmd[, walltime, envs])	Synchronously execute a commandline string on the shell.
isdir(path)	Return true if the path refers to an existing directory.
makedirs(path[, mode, exist_ok])	Create a directory on the remote side.
prepend_envs(cmd[, env])	
pull_file(remote_source, local_dir)	Transport file on the remote side to a local directory
push_file(local_source, remote_dir)	Transport a local file to a directory on a remote machine

Attributes

script_dir	This is a property.

5.22 parsl.channels.OAuthSSHChannel

class parsl.channels.**OAuthSSHChannel**(*hostname*, *username=None*, *script_dir=None*, *envs=None*, *port=22*)

SSH persistent channel. This enables remote execution on sites accessible via ssh. This channel uses Globus based OAuth tokens for authentication.

__init__(*hostname*, *username=None*, *script_dir=None*, *envs=None*, *port=22*)

Initialize a persistent connection to the remote system. We should know at this point whether ssh connectivity is possible

Parameters **hostname** (*-*) – Hostname

KWargs:

- username (string) : Username on remote system
- script_dir (string) : Full path to a script dir where generated scripts could be sent to.
- envs (dict) : A dictionary of env variables to be set when executing commands
- port (int) : Port at which the SSHService is running

Raises:

Methods

`__init__`(hostname[, username, script_dir, . . .])	Initialize a persistent connection to the remote system.
`abspath`(path)	Return the absolute path on the remote side.
`close`()	Closes the channel.
`execute_no_wait`(cmd[, walltime, envs])	Execute asynchronousely without waiting for exitcode
`execute_wait`(cmd[, walltime, envs])	Synchronously execute a commandline string on the shell.
`isdir`(path)	Return true if the path refers to an existing directory.
`makedirs`(path[, mode, exist_ok])	Create a directory on the remote side.
`prepend_envs`(cmd[, env])	
`pull_file`(remote_source, local_dir)	Transport file on the remote side to a local directory
`push_file`(local_source, remote_dir)	Transport a local file to a directory on a remote machine

Attributes

`script_dir`	This is a property.

5.23 parsl.channels.SSHInteractiveLoginChannel

class parsl.channels.**SSHInteractiveLoginChannel**(*hostname*, *username=None*, *password=None*, *script_dir=None*, *envs=None*, ***kwargs*)

SSH persistent channel. This enables remote execution on sites accessible via ssh. This channel supports interactive login and is appropriate when keys are not set up.

`__init__`(*hostname*, *username=None*, *password=None*, *script_dir=None*, *envs=None*, ***kwargs*)
Initialize a persistent connection to the remote system. We should know at this point whether ssh connectivity is possible

Parameters **`hostname`** (*–*) – Hostname

KWargs:

- username (string) : Username on remote system
- password (string) : Password for remote system
- script_dir (string) : Full path to a script dir where generated scripts could be sent to.
- envs (dict) : A dictionary of env variables to be set when executing commands

Raises:

Methods

`__init__`(hostname[, username, password, . . .])	Initialize a persistent connection to the remote system.
`abspath`(path)	Return the absolute path on the remote side.
`close`()	Closes the channel.
`execute_no_wait`(cmd[, walltime, envs])	Execute asynchronousely without waiting for exitcode
`execute_wait`(cmd[, walltime, envs])	Synchronously execute a commandline string on the shell.
`isdir`(path)	Return true if the path refers to an existing directory.
`makedirs`(path[, mode, exist_ok])	Create a directory on the remote side.
`prepend_envs`(cmd[, env])	
`pull_file`(remote_source, local_dir)	Transport file on the remote side to a local directory
`push_file`(local_source, remote_dir)	Transport a local file to a directory on a remote machine

Attributes

`script_dir`	This is a property.

5.24 parsl.providers.AdHocProvider

class `parsl.providers.`**`AdHocProvider`**(*channels=[]*, *worker_init=”*, *cmd_timeout=30*, *parallelism=1*, *move_files=None*)
Ad-hoc execution provider

This provider is used to provision execution resources over one or more ad hoc nodes that are each accessible over a Channel (say, ssh) but otherwise lack a cluster scheduler.

Parameters

- **`channels`** (*`list of Channel ojects`*) – Each channel represents a connection to a remote node

- **worker_init** (*str*) – Command to be run before starting a worker, such as 'module load Anaconda; source activate env'. Since this provider calls the same worker_init across all nodes in the ad-hoc cluster, it is recommended that a single script is made available across nodes such as ~/setup.sh that can be invoked.
- **cmd_timeout** (*int*) – Duration for which the provider will wait for a command to be invoked on a remote system. Defaults to 30s
- **parallelism** (*float*) – Determines the ratio of workers to tasks as managed by the strategy component

__init__(*channels=[]*, *worker_init="*, *cmd_timeout=30*, *parallelism=1*, *move_files=None*)
Initialize self. See help(type(self)) for accurate signature.

Methods

`__init__`([channels, worker_init, . . .])	Initialize self.
`cancel`(job_ids)	Cancel a list of jobs with job_ids
`status`(job_ids)	Get status of the list of jobs with job_ids
`submit`(command, tasks_per_node[, job_name])	Submits the command onto a channel from the list of channels

Attributes

`cores_per_node`	Number of cores to provision per node.
`label`	Provides the label for this provider
`mem_per_node`	Real memory to provision per node in GB.
`scaling_enabled`	

5.25 parsl.providers.AWSProvider

class parsl.providers.**AWSProvider**(*image_id*, *key_name*, *init_blocks=1*, *min_blocks=0*, *max_blocks=10*, *nodes_per_block=1*, *parallelism=1*, *worker_init="*, *instance_type='t2.small'*, *region='us-east-2'*, *spot_max_bid=0*, *key_file=None*, *profile=None*, *iam_instance_profile_arn="*, *state_file=None*, *walltime='01:00:00'*, *linger=False*, *launcher=SingleNodeLauncher()*)
A provider for using Amazon Elastic Compute Cloud (EC2) resources.

One of 3 methods are required to authenticate: keyfile, profile or environment variables. If neither keyfile or profile are set, the following environment variables must be set: `AWS_ACCESS_KEY_ID` (the access key for your AWS account), `AWS_SECRET_ACCESS_KEY` (the secret key for your AWS account), and (optionaly) the `AWS_SESSION_TOKEN` (the session key for your AWS account).

Parameters

- **image_id** (*str*) – Identification of the Amazon Machine Image (AMI).
- **worker_init** (*str*) – String to append to the Userdata script executed in the cloudinit phase of instance initialization.
- **walltime** (*str*) – Walltime requested per block in HH:MM:SS.

- **key_file** (*str*) – Path to json file that contains 'AWSAccessKeyId' and 'AWSSecretKey'.
- **nodes_per_block** (*int*) – This is always 1 for ec2. Nodes to provision per block.
- **profile** (*str*) – Profile to be used from the standard aws config file ~/.aws/config.
- **nodes_per_block** – Nodes to provision per block. Default is 1.
- **init_blocks** (*int*) – Number of blocks to provision at the start of the run. Default is 1.
- **min_blocks** (*int*) – Minimum number of blocks to maintain. Default is 0.
- **max_blocks** (*int*) – Maximum number of blocks to maintain. Default is 10.
- **instance_type** (*str*) – EC2 instance type. Instance types comprise varying combinations of CPU, memory, . storage, and networking capacity For more information on possible instance types,. see here Default is 't2.small'.
- **region** (*str*) – Amazon Web Service (AWS) region to launch machines. Default is 'us-east-2'.
- **key_name** (*str*) – Name of the AWS private key (.pem file) that is usually generated on the console to allow SSH access to the EC2 instances. This is mostly used for debugging.
- **spot_max_bid** (*float*) – Maximum bid price (if requesting spot market machines).
- **iam_instance_profile_arn** (*str*) – Launch instance with a specific role.
- **state_file** (*str*) – Path to the state file from a previous run to re-use.
- **walltime** – Walltime requested per block in HH:MM:SS. This option is not currently honored by this provider.
- **launcher** (*Launcher*) – Launcher for this provider. Possible launchers include *SingleNodeLauncher* (the default), *SrunLauncher*, or *AprunLauncher*
- **linger** (*Bool*) – When set to True, the workers will not `halt`. The user is responsible for shutting down the node.

__init__(*image_id, key_name, init_blocks=1, min_blocks=0, max_blocks=10, nodes_per_block=1, parallelism=1, worker_init=", instance_type='t2.small', region='us-east-2', spot_max_bid=0, key_file=None, profile=None, iam_instance_profile_arn=", state_file=None, walltime='01:00:00', linger=False, launcher=SingleNodeLauncher()*)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(image_id, key_name[, init_blocks, . . .])	Initialize self.
cancel(job_ids)	Cancel the jobs specified by a list of job ids.
config_route_table(vpc, internet_gateway)	Configure route table for Virtual Private Cloud (VPC).
create_session()	Create a session.
create_vpc()	Create and configure VPC
get_instance_state([instances])	Get states of all instances on EC2 which were started by this file.
goodbye()	
initialize_boto_client()	Initialize the boto client.
read_state_file(state_file)	Read the state file, if it exists.

Continued on next page

Table 40 – continued from previous page

security_group(vpc)	Create and configure a new security group.
show_summary()	Print human readable summary of current AWS state to log and to console.
shut_down_instance([instances])	Shut down a list of instances, if provided.
spin_up_instance(command, job_name)	Start an instance in the VPC in the first available subnet.
status(job_ids)	Get the status of a list of jobs identified by their ids.
submit([command, tasks_per_node, job_name])	Submit the command onto a freshly instantiated AWS EC2 instance.
teardown()	Teardown the EC2 infastructure.
write_state_file()	Save information that must persist to a file.
xstr(s)	

Attributes

current_capacity	Returns the current blocksize.
label	Provides the label for this provider
AWSProvider.scaling_enabled	

5.26 parsl.providers.CobaltProvider

class parsl.providers.**CobaltProvider**(*channel=LocalChannel(envs={}, script_dir=None, userhome='/home/docs/checkouts/readthedocs.org/user_builds/parsl/checkouts/0.9.0), nodes_per_block=1, init_blocks=0, min_blocks=0, max_blocks=10, parallelism=1, walltime='00:10:00', account=None, queue=None, scheduler_options='', worker_init='', launcher=AprunLauncher(overrides=''), cmd_timeout=10*)

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.

Parameters

- **channel** (*Channel*) – Channel for accessing this provider. Possible channels include *LocalChannel* (the default), *SSHChannel*, or *SSHInteractiveLoginChannel*.
- **nodes_per_block** (*int*) – Nodes to provision per block.
- **min_blocks** (*int*) – Minimum number of blocks to maintain.
- **max_blocks** (*int*) – Maximum number of blocks to maintain.
- **walltime** (*str*) – Walltime requested per block in HH:MM:SS.
- **account** (*str*) – Account that the job will be charged against.
- **queue** (*str*) – Torque queue to request blocks from.
- **scheduler_options** (*str*) – String to prepend to the submit script to the scheduler.
- **worker_init** (*str*) – Command to be run before starting a worker, such as 'module load Anaconda; source activate env'.

- **launcher** (*Launcher*) – Launcher for this provider. Possible launchers include *AprunLauncher* (the default) or, *SingleNodeLauncher*

__init__(*channel=LocalChannel(envs={}, script_dir=None, userhome='/home/docs/checkouts/readthedocs.org/user_builds/parsl/checkouts/0.9.0/docs'), nodes_per_block=1, init_blocks=0, min_blocks=0, max_blocks=10, parallelism=1, walltime='00:10:00', account=None, queue=None, scheduler_options='', worker_init='', launcher=AprunLauncher(overrides=''), cmd_timeout=10*)
Initialize self. See help(type(self)) for accurate signature.

Methods

`__init__`([channel, nodes_per_block, ...])	Initialize self.
`cancel`(job_ids)	Cancels the jobs specified by a list of job ids
`execute_wait`(cmd[, timeout])	
`status`(job_ids)	Get the status of a list of jobs identified by the job identifiers returned from the submit request.
`submit`(command, tasks_per_node[, job_name])	Submits the command onto an Local Resource Manager job of parallel elements.

Attributes

`current_capacity`	Returns the currently provisioned blocks.
`label`	Provides the label for this provider
`CobaltProvider.scaling_enabled`	

5.27 parsl.providers.CondorProvider

class parsl.providers.**CondorProvider**(*channel: parsl.channels.base.Channel = LocalChannel(envs={}, script_dir=None, userhome='/home/docs/checkouts/readthedocs.org/user_builds/parsl/checkouts/0.9.0), nodes_per_block: int = 1, cores_per_slot: Optional[int] = None, mem_per_slot: Optional[float] = None, init_blocks: int = 1, min_blocks: int = 0, max_blocks: int = 10, parallelism: float = 1, environment: Optional[Dict[str, str]] = None, project: str = '', scheduler_options: str = '', transfer_input_files: List[str] = [], walltime: str = '00:10:00', worker_init: str = '', launcher: parsl.launchers.launchers.Launcher = SingleNodeLauncher(), requirements: str = '', cmd_timeout: int = 60*)

HTCondor Execution Provider.

Parameters

- **channel** (Channel) – Channel for accessing this provider. Possible channels include *LocalChannel* (the default), *SSHChannel*, or *SSHInteractiveLoginChannel*.
- **nodes_per_block** (*int*) – Nodes to provision per block.
- **cores_per_slot** (*int*) – Specify the number of cores to provision per slot. If set to None, executors will assume all cores on the node are available for computation. Default is None.

- **mem_per_slot** (*float*) – Specify the real memory to provision per slot in GB. If set to None, no explicit request to the scheduler will be made. Default is None.
- **init_blocks** (*int*) – Number of blocks to provision at time of initialization
- **min_blocks** (*int*) – Minimum number of blocks to maintain
- **max_blocks** (*int*) – Maximum number of blocks to maintain.
- **parallelism** (*float*) – Ratio of provisioned task slots to active tasks. 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., min_blocks) are used.
- **environment** (*dict of str*) – A dictionary of environmant variable name and value pairs which will be set before running a task.
- **project** (*str*) – Project which the job will be charged against
- **scheduler_options** (*str*) – String to add specific condor attributes to the HTCondor submit script.
- **transfer_input_files** (*list (str)*) – List of strings of paths to additional files or directories to transfer to the job
- **worker_init** (*str*) – Command to be run before starting a worker.
- **requirements** (*str*) – Condor requirements.
- **launcher** (*Launcher*) – Launcher for this provider. Possible launchers include *SingleNodeLauncher* (the default),
- **cmd_timeout** (*int*) – Timeout for commands made to the scheduler in seconds

__init__(*channel: parsl.channels.base.Channel = LocalChannel(envs={}, script_dir=None, userhome='/home/docs/checkouts/readthedocs.org/user_builds/parsl/checkouts/0.9.0/docs'), nodes_per_block: int = 1, cores_per_slot: Optional[int] = None, mem_per_slot: Optional[float] = None, init_blocks: int = 1, min_blocks: int = 0, max_blocks: int = 10, parallelism: float = 1, environment: Optional[Dict[str, str]] = None, project: str = '', scheduler_options: str = '', transfer_input_files: List[str] = [], walltime: str = '00:10:00', worker_init: str = '', launcher: parsl.launchers.launchers.Launcher = SingleNodeLauncher(), requirements: str = '', cmd_timeout: int = 60*) → None

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

Methods

__init__(channel, script_dir=None, . . .)	Initialize self.
cancel(job_ids)	Cancels the jobs specified by a list of job IDs.
execute_wait(cmd[, timeout])	
status(job_ids)	Get the status of a list of jobs identified by their ids.
submit(command, tasks_per_node[, job_name])	Submits the command onto an Local Resource Manager job.

Attributes

current_capacity	Returns the currently provisioned blocks.
label	Provides the label for this provider

Continued on next page

Table 45 – continued from previous page

CondorProvider.scaling_enabled

5.28 parsl.providers.GoogleCloudProvider

class parsl.providers.**GoogleCloudProvider**(*project_id*, *key_file*, *region*, *os_project*, *os_family*, *google_version='v1'*, *instance_type='n1-standard-1'*, *init_blocks=1*, *min_blocks=0*, *max_blocks=10*, *launcher=SingleNodeLauncher()*, *parallelism=1*)

A provider for using resources from the Google Compute Engine.

Parameters

- **project_id** (*str*) – Project ID from Google compute engine.
- **key_file** (*str*) – Path to authorization private key json file. This is required for auth. A new one can be generated here: https://console.cloud.google.com/apis/credentials
- **region** (*str*) – Region in which to start instances
- **os_project** (*str*) – OS project code for Google compute engine.
- **os_family** (*str*) – OS family to request.
- **google_version** (*str*) – Google compute engine version to use. Possibilies include 'v1' (default) or 'beta'.
- **instance_type** (*str*) – 'n1-standard-1',
- **init_blocks** (*int*) – Number of blocks to provision immediately. Default is 1.
- **min_blocks** (*int*) – Minimum number of blocks to maintain. Default is 0.
- **max_blocks** (*int*) – Maximum number of blocks to maintain. Default is 10.
- **parallelism** (*float*) – Ratio of provisioned task slots to active tasks. 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., min_blocks) are used.

:param .. code:: python: +——————

script_string ——->| submit

id <———|—+

[ids] ——->| status [statuses] <———|—-+

[ids] ——->| cancel [cancel] <———|—-+

+——————-

__init__(*project_id, key_file, region, os_project, os_family, google_version='v1', instance_type='n1-standard-1', init_blocks=1, min_blocks=0, max_blocks=10, launcher=SingleNodeLauncher(), parallelism=1*)

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

Methods

`__init__`(project_id, key_file, region, ...)	Initialize self.
`bye`()	
`cancel`(job_ids)	Cancels the resources identified by the job_ids provided by the user.
`create_instance`([command])	
`delete_instance`(name)	
`get_zone`(region)	
`status`(job_ids)	Get the status of a list of jobs identified by the job identifiers returned from the submit request.
`submit`(command, tasks_per_node[, job_name])	The submit method takes the command string to be executed upon instantiation of a resource most often to start a pilot.

Attributes

`current_capacity`	Returns the number of currently provisioned blocks.
`GoogleCloudProvider.scaling_enabled`	

5.29 parsl.providers.GridEngineProvider

class parsl.providers.**GridEngineProvider**(*channel=LocalChannel(envs={}, script_dir=None, userhome='/home/docs/checkouts/readthedocs.org/user_builds/parsl/checkout), nodes_per_block=1, init_blocks=1, min_blocks=0, max_blocks=10, parallelism=1, walltime='00:10:00', scheduler_options='', worker_init='', launcher=SingleNodeLauncher()*)

A provider for the Grid Engine scheduler.

Parameters

- **channel** (`Channel`) – Channel for accessing this provider. Possible channels include `LocalChannel` (the default), `SSHChannel`, or `SSHInteractiveLoginChannel`.
- **nodes_per_block** (`int`) – Nodes to provision per block.
- **min_blocks** (`int`) – Minimum number of blocks to maintain.
- **max_blocks** (`int`) – Maximum number of blocks to maintain.
- **parallelism** (`float`) – Ratio of provisioned task slots to active tasks. 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., min_blocks) are used.

- **walltime** (*str*) – Walltime requested per block in HH:MM:SS.
- **scheduler_options** (*str*) – String to prepend to the #$$ blocks in the submit script to the scheduler.
- **worker_init** (*str*) – Command to be run before starting a worker, such as 'module load Anaconda; source activate env'.
- **launcher** (*Launcher*) – Launcher for this provider. Possible launchers include *SingleNodeLauncher* (the default),

__init__(*channel=LocalChannel(envs={}, script_dir=None, user-home='/home/docs/checkouts/readthedocs.org/user_builds/parsl/checkouts/0.9.0/docs'), nodes_per_block=1, init_blocks=1, min_blocks=0, max_blocks=10, parallelism=1, walltime='00:10:00', scheduler_options='', worker_init='', launcher=SingleNodeLauncher()*)
Initialize self. See help(type(self)) for accurate signature.

Methods

`__init__`([channel, nodes_per_block, ...])	Initialize self.
`cancel`(job_ids)	Cancels the resources identified by the job_ids provided by the user.
`execute_wait`(cmd[, timeout])	
`get_configs`(command, tasks_per_node)	Compose a dictionary with information for writing the submit script.
`status`(job_ids)	Get the status of a list of jobs identified by the job identifiers returned from the submit request.
`submit`(command, tasks_per_node[, job_name])	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).

Attributes

`current_capacity`	Returns the currently provisioned blocks.
`label`	Provides the label for this provider
`GridEngineProvider.scaling_enabled`	

5.30 parsl.providers.JetstreamProvider

class parsl.providers.**JetstreamProvider**(*config, poolname*)

__init__(*config, poolname*)
Initialize self. See help(type(self)) for accurate signature.

Methods

`__init__`(config, poolname)	Initialize self.
`scale_in`([blocks, machines, strategy])	Scale in resources

Continued on next page

Table 50 – continued from previous page

`scale_out`([blocks, block_size])	Scale out the existing resources.

5.31 parsl.providers.LocalProvider

class parsl.providers.**LocalProvider**(*channel=LocalChannel(envs={}, script_dir=None, user-home='/home/docs/checkouts/readthedocs.org/user_builds/parsl/checkouts/0.9.0/), nodes_per_block=1, launcher=SingleNodeLauncher(), init_blocks=4, min_blocks=0, max_blocks=10, walltime='00:15:00', worker_init='', cmd_timeout=30, parallelism=1, move_files=None*)

Local Execution Provider

This provider is used to provide execution resources from the localhost.

Parameters

- **min_blocks** (*int*) – Minimum number of blocks to maintain.
- **max_blocks** (*int*) – Maximum number of blocks to maintain.
- **parallelism** (*float*) – Ratio of provisioned task slots to active tasks. 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., min_blocks) are used.
- **move_files** (*Optional[Bool]: should files be moved? by default, Parsl will try to figure*) – this out itself (= None). If True, then will always move. If False, will never move.
- **worker_init** (*str*) – Command to be run before starting a worker, such as 'module load Anaconda; source activate env'.

__init__(*channel=LocalChannel(envs={}, script_dir=None, user-home='/home/docs/checkouts/readthedocs.org/user_builds/parsl/checkouts/0.9.0/docs'), nodes_per_block=1, launcher=SingleNodeLauncher(), init_blocks=4, min_blocks=0, max_blocks=10, walltime='00:15:00', worker_init='', cmd_timeout=30, parallelism=1, move_files=None*)

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

Methods

`__init__`([channel, nodes_per_block, ...])	Initialize self.
`cancel`(job_ids)	Cancels the jobs specified by a list of job ids
`status`(job_ids)	Get the status of a list of jobs identified by their ids.
`submit`(command, tasks_per_node[, job_name])	Submits the command onto an Local Resource Manager job.

Attributes

`current_capacity`	
`label`	Provides the label for this provider

Continued on next page

Table 52 – continued from previous page

`LocalProvider.scaling_enabled`

5.32 parsl.providers.LSFProvider

class parsl.providers.**LSFProvider**(*channel=LocalChannel(envs={}, script_dir=None, userhome='/home/docs/checkouts/readthedocs.org/user_builds/parsl/checkouts/0.9.0/doc), nodes_per_block=1, init_blocks=1, min_blocks=0, max_blocks=10, parallelism=1, walltime='00:10:00', scheduler_options='', worker_init='', project=None, cmd_timeout=120, move_files=True, launcher=SingleNodeLauncher()*)

LSF 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.

Parameters

- **channel** (`Channel`) – Channel for accessing this provider. Possible channels include `LocalChannel` (the default), `SSHChannel`, or `SSHInteractiveLoginChannel`.
- **nodes_per_block** (`int`) – Nodes to provision per block.
- **init_blocks** (`int`) – Number of blocks to request at the start of the run.
- **min_blocks** (`int`) – Minimum number of blocks to maintain.
- **max_blocks** (`int`) – Maximum number of blocks to maintain.
- **parallelism** (`float`) – Ratio of provisioned task slots to active tasks. 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., min_blocks) are used.
- **walltime** (`str`) – Walltime requested per block in HH:MM:SS.
- **project** (`str`) – Project to which the resources must be charged
- **scheduler_options** (`str`) – String to prepend to the #SBATCH blocks in the submit script to the scheduler.
- **worker_init** (`str`) – Command to be run before starting a worker, such as 'module load Anaconda; source activate env'.
- **cmd_timeout** (`int`) – Seconds after which requests to the scheduler will timeout. Default: 120s
- **launcher** (`Launcher`) –

  Launcher for this provider. Possible launchers include `SingleNodeLauncher` (the default), `SrunLauncher`, or `AprunLauncher`

  move_files : Optional[Bool]: should files be moved? by default, Parsl will try to move files.

__init__(*channel=LocalChannel(envs={}, script_dir=None, userhome='/home/docs/checkouts/readthedocs.org/user_builds/parsl/checkouts/0.9.0/docs'), nodes_per_block=1, init_blocks=1, min_blocks=0, max_blocks=10, parallelism=1, walltime='00:10:00', scheduler_options='', worker_init='', project=None, cmd_timeout=120, move_files=True, launcher=SingleNodeLauncher()*)

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

Methods

`__init__`([channel, nodes_per_block, ...])	Initialize self.
`cancel`(job_ids)	Cancels the jobs specified by a list of job ids
`execute_wait`(cmd[, timeout])	
`status`(job_ids)	Get the status of a list of jobs identified by the job identifiers returned from the submit request.
`submit`(command, tasks_per_node[, job_name])	Submit the command as an LSF job.

Attributes

`cores_per_node`	Number of cores to provision per node.
`current_capacity`	Returns the currently provisioned blocks.
`label`	Provides the label for this provider
`mem_per_node`	Real memory to provision per node in GB.

5.33 parsl.providers.SlurmProvider

class parsl.providers.**SlurmProvider**(*partition, channel=LocalChannel(envs={}, script_dir=None, user-home='/home/docs/checkouts/readthedocs.org/user_builds/parsl/checkouts/0.9.0/), nodes_per_block=1, cores_per_node=None, mem_per_node=None, init_blocks=1, min_blocks=0, max_blocks=10, parallelism=1, walltime='00:10:00', scheduler_options='', worker_init='', cmd_timeout=10, exclusive=True, move_files=True, launcher=SingleNodeLauncher()*)

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.

Parameters

- **partition** (*str*) – Slurm partition to request blocks from.
- **channel** (`Channel`) – Channel for accessing this provider. Possible channels include *LocalChannel* (the default), *SSHChannel*, or *SSHInteractiveLoginChannel*.
- **nodes_per_block** (*int*) – Nodes to provision per block.
- **cores_per_node** (*int*) – Specify the number of cores to provision per node. If set to None, executors will assume all cores on the node are available for computation. Default is None.
- **mem_per_node** (*float*) – Specify the real memory to provision per node in GB. If set to None, no explicit request to the scheduler will be made. Default is None.
- **min_blocks** (*int*) – Minimum number of blocks to maintain.
- **max_blocks** (*int*) – Maximum number of blocks to maintain.
- **parallelism** (*float*) – Ratio of provisioned task slots to active tasks. 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., min_blocks) are used.

- **walltime** (*str*) – Walltime requested per block in HH:MM:SS.
- **scheduler_options** (*str*) – String to prepend to the #SBATCH blocks in the submit script to the scheduler.
- **worker_init** (*str*) – Command to be run before starting a worker, such as 'module load Anaconda; source activate env'.
- **exclusive** (*bool (Default = True)*) – Requests nodes which are not shared with other running jobs.
- **launcher** (*Launcher*) –

  Launcher for this provider. Possible launchers include `SingleNodeLauncher` (the default), `SrunLauncher`, or `AprunLauncher`

  move_files : Optional[Bool]: should files be moved? by default, Parsl will try to move files.

__init__(*partition, channel=LocalChannel(envs={}, script_dir=None, userhome='/home/docs/checkouts/readthedocs.org/user_builds/parsl/checkouts/0.9.0/docs'), nodes_per_block=1, cores_per_node=None, mem_per_node=None, init_blocks=1, min_blocks=0, max_blocks=10, parallelism=1, walltime='00:10:00', scheduler_options='', worker_init='', cmd_timeout=10, exclusive=True, move_files=True, launcher=SingleNodeLauncher()*)
Initialize self. See help(type(self)) for accurate signature.

Methods

`__init__`(partition[, channel, . . .])	Initialize self.
`cancel`(job_ids)	Cancels the jobs specified by a list of job ids
`execute_wait`(cmd[, timeout])	
`status`(job_ids)	Get the status of a list of jobs identified by the job identifiers returned from the submit request.
`submit`(command, tasks_per_node[, job_name])	Submit the command as a slurm job.

Attributes

`current_capacity`	Returns the currently provisioned blocks.
`label`	Provides the label for this provider
`SlurmProvider.scaling_enabled`	

5.34 parsl.providers.TorqueProvider

class parsl.providers.**TorqueProvider**(*channel=LocalChannel(envs={}, script_dir=None, userhome='/home/docs/checkouts/readthedocs.org/user_builds/parsl/checkouts/0.9.( ), account=None, queue=None, scheduler_options='', worker_init='', nodes_per_block=1, init_blocks=1, min_blocks=0, max_blocks=100, parallelism=1, launcher=AprunLauncher(overrides=''), walltime='00:20:00', cmd_timeout=120*)

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.

Parameters

- **channel** (*Channel*) – Channel for accessing this provider. Possible channels include *LocalChannel* (the default), *SSHChannel*, or *SSHInteractiveLoginChannel*.
- **account** (*str*) – Account the job will be charged against.
- **queue** (*str*) – Torque queue to request blocks from.
- **nodes_per_block** (*int*) – Nodes to provision per block.
- **init_blocks** (*int*) – Number of blocks to provision at the start of the run. Default is 1.
- **min_blocks** (*int*) – Minimum number of blocks to maintain. Default is 0.
- **max_blocks** (*int*) – Maximum number of blocks to maintain.
- **parallelism** (*float*) – Ratio of provisioned task slots to active tasks. 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., min_blocks) are used.
- **walltime** (*str*) – Walltime requested per block in HH:MM:SS.
- **scheduler_options** (*str*) – String to prepend to the #PBS blocks in the submit script to the scheduler.
- **worker_init** (*str*) – Command to be run before starting a worker, such as 'module load Anaconda; source activate env'.
- **launcher** (*Launcher*) – Launcher for this provider. Possible launchers include *AprunLauncher* (the default), or *SingleNodeLauncher*,

__init__(*channel=LocalChannel(envs={}, script_dir=None, userhome='/home/docs/checkouts/readthedocs.org/user_builds/parsl/checkouts/0.9.0/docs'), account=None, queue=None, scheduler_options='', worker_init='', nodes_per_block=1, init_blocks=1, min_blocks=0, max_blocks=100, parallelism=1, launcher=AprunLauncher(overrides=''), walltime='00:20:00', cmd_timeout=120*)

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

Methods

`__init__`([channel, account, queue, ...])	Initialize self.
`cancel`(job_ids)	Cancels the jobs specified by a list of job ids
`execute_wait`(cmd[, timeout])	
`status`(job_ids)	Get the status of a list of jobs identified by the job identifiers returned from the submit request.
`submit`(command, tasks_per_node[, job_name])	Submits the command onto an Local Resource Manager job.

Attributes

`current_capacity`	Returns the currently provisioned blocks.
`label`	Provides the label for this provider
`TorqueProvider.scaling_enabled`	

5.35 parsl.providers.KubernetesProvider

class parsl.providers.**KubernetesProvider**(*image: str, namespace: str = 'default', nodes_per_block: int = 1, init_blocks: int = 4, min_blocks: int = 0, max_blocks: int = 10, max_cpu: float = 2, max_mem: str = '500Mi', init_cpu: float = 1, init_mem: str = '250Mi', parallelism: float = 1, worker_init: str = '', pod_name: Optional[str] = None, user_id: Optional[str] = None, group_id: Optional[str] = None, run_as_non_root: bool = False, secret: Optional[str] = None, persistent_volumes: List[Tuple[str, str]] = []*)

Kubernetes execution provider :param namespace: Kubernetes namespace to create deployments. :type namespace: str :param image: Docker image to use in the deployment. :type image: str :param nodes_per_block: Nodes to provision per block. :type nodes_per_block: int :param init_blocks: Number of blocks to provision at the start of the run. Default is 1. :type init_blocks: int :param min_blocks: Minimum number of blocks to maintain. :type min_blocks: int :param max_blocks: Maximum number of blocks to maintain. :type max_blocks: int :param max_cpu: CPU limits of the blocks (pods), in cpu units.

This is the cpu "limits" option for resource specification. Check kubernetes docs for more details. Default is 2.

Parameters

- **max_mem** (*str*) – Memory limits of the blocks (pods), in Mi or Gi. This is the memory "limits" option for resource specification on kubernetes. Check kubernetes docs for more details. Default is 500Mi.
- **init_cpu** (*float*) – CPU limits of the blocks (pods), in cpu units. This is the cpu "requests" option for resource specification. Check kubernetes docs for more details. Default is 1.
- **init_mem** (*str*) – Memory limits of the blocks (pods), in Mi or Gi. This is the memory "requests" option for resource specification on kubernetes. Check kubernetes docs for more details. Default is 250Mi.
- **parallelism** (*float*) – Ratio of provisioned task slots to active tasks. 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., min_blocks) are used.
- **worker_init** (*str*) – Command to be run first for the workers, such as `python start.py`.
- **secret** (*str*) – Docker secret to use to pull images
- **pod_name** (*str*) – The name for the pod, will be appended with a timestamp. Default is None, meaning parsl automatically names the pod.
- **user_id** (*str*) – Unix user id to run the container as.
- **group_id** (*str*) – Unix group id to run the container as.
- **run_as_non_root** (*bool*) – Run as non-root (True) or run as root (False).
- **persistent_volumes** (*list[(str, str)]*) – List of tuples describing persistent volumes to be mounted in the pod. The tuples consist of (PVC Name, Mount Directory).

__init__(*image: str*, *namespace: str = 'default'*, *nodes_per_block: int = 1*, *init_blocks: int = 4*, *min_blocks: int = 0*, *max_blocks: int = 10*, *max_cpu: float = 2*, *max_mem: str = '500Mi'*, *init_cpu: float = 1*, *init_mem: str = '250Mi'*, *parallelism: float = 1*, *worker_init: str = ''*, *pod_name: Optional[str] = None*, *user_id: Optional[str] = None*, *group_id: Optional[str] = None*, *run_as_non_root: bool = False*, *secret: Optional[str] = None*, *persistent_volumes: List[Tuple[str, str]] = []*) → None

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

Methods

`__init__`(image, namespace, nodes_per_block, ...)	Initialize 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.
`status`(job_ids)	Get the status of a list of jobs identified by the job identifiers returned from the submit request.
`submit`(cmd_string, tasks_per_node[, job_name])	Submit a job :param - cmd_string: (String) - Name of the container to initiate :param - tasks_per_node: command invocations to be launched per node :type - tasks_per_node: int

Attributes

`KubernetesProvider.channels_required`	
`label`	Provides the label for this provider
`KubernetesProvider.scaling_enabled`	

5.36 parsl.providers.PBSProProvider

class parsl.providers.**PBSProProvider**(*channel=LocalChannel(envs={}, script_dir=None, userhome='/home/docs/checkouts/readthedocs.org/user_builds/parsl/checkouts/0.9.0), account=None, queue=None, scheduler_options='', worker_init='', nodes_per_block=1, cpus_per_node=1, init_blocks=1, min_blocks=0, max_blocks=100, parallelism=1, launcher=SingleNodeLauncher(), walltime='00:20:00', cmd_timeout=120*)

PBS Pro 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.

Parameters

- **channel** (`Channel`) – Channel for accessing this provider. Possible channels include `LocalChannel` (the default), `SSHChannel`, or `SSHInteractiveLoginChannel`.
- **account** (`str`) – Account the job will be charged against.

- **queue** (*str*) – Queue to request blocks from.
- **nodes_per_block** (*int*) – Nodes to provision per block.
- **cpus_per_node** (*int*) – CPUs to provision per node.
- **init_blocks** (*int*) – Number of blocks to provision at the start of the run. Default is 1.
- **min_blocks** (*int*) – Minimum number of blocks to maintain. Default is 0.
- **max_blocks** (*int*) – Maximum number of blocks to maintain.
- **parallelism** (*float*) – Ratio of provisioned task slots to active tasks. 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., min_blocks) are used.
- **walltime** (*str*) – Walltime requested per block in HH:MM:SS.
- **scheduler_options** (*str*) – String to prepend to the #PBS blocks in the submit script to the scheduler.
- **worker_init** (*str*) – Command to be run before starting a worker, such as 'module load Anaconda; source activate env'.
- **launcher** (*Launcher*) – Launcher for this provider. The default is *SingleNodeLauncher*.

__init__(*channel=LocalChannel(envs={}, script_dir=None, userhome='/home/docs/checkouts/readthedocs.org/user_builds/parsl/checkouts/0.9.0/docs'), account=None, queue=None, scheduler_options='', worker_init='', nodes_per_block=1, cpus_per_node=1, init_blocks=1, min_blocks=0, max_blocks=100, parallelism=1, launcher=SingleNodeLauncher(), walltime='00:20:00', cmd_timeout=120*)

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

Methods

`__init__`([channel, account, queue, . . .])	Initialize self.
`cancel`(job_ids)	Cancels the jobs specified by a list of job ids
`execute_wait`(cmd[, timeout])	
`status`(job_ids)	Get the status of a list of jobs identified by the job identifiers returned from the submit request.
`submit`(command, tasks_per_node[, job_name])	Submits the command job.

Attributes

`cores_per_node`	Number of cores to provision per node.
`current_capacity`	Returns the currently provisioned blocks.
`label`	Provides the label for this provider
`mem_per_node`	Real memory to provision per node in GB.
`PBSProProvider.scaling_enabled`	

5.37 parsl.launchers.SimpleLauncher

class `parsl.launchers.`**`SimpleLauncher`**

Does no wrapping. Just returns the command as-is

`__init__`()

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

5.38 parsl.launchers.SingleNodeLauncher

class `parsl.launchers.`**`SingleNodeLauncher`**

Worker launcher that wraps the user's command with the framework to launch multiple command invocations in parallel. This wrapper sets the bash env variable CORES to the number of cores on the machine. By setting task_blocks to an integer or to a bash expression the number of invocations of the command to be launched can be controlled.

`__init__`()

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

5.39 parsl.launchers.SrunLauncher

class `parsl.launchers.`**`SrunLauncher`**(*overrides=''*)

Worker launcher that wraps the user's command with the SRUN launch framework to launch multiple cmd invocations in parallel on a single job allocation.

`__init__`(*overrides=''*)

Parameters **`overrides`** (*`str`*) – This string will be passed to the srun launcher. Default: ''

Methods

`__init__`([overrides])	**param overrides** This string will be passed to the srun launcher. Default: ''

5.40 parsl.launchers.AprunLauncher

class `parsl.launchers.`**`AprunLauncher`**(*overrides=''*)

Worker launcher that wraps the user's command with the Aprun launch framework to launch multiple cmd invocations in parallel on a single job allocation

`__init__`(*overrides=''*)

Parameters **`overrides`** (*`str`*) – This string will be passed to the aprun launcher. Default: ''

Methods

__init__([overrides])

param overrides This string will be passed to the aprun launcher. Default: ‘’

5.41 parsl.launchers.SrunMPILauncher

class parsl.launchers.**SrunMPILauncher**(*overrides=”*)

Launches as many workers as MPI tasks to be executed concurrently within a block.

Use this launcher instead of SrunLauncher if each block will execute multiple MPI applications at the same time. Workers should be launched with independent Srun calls so as to setup the environment for MPI application launch.

__init__(*overrides=”*)

Parameters **overrides** (*str*) – This string will be passed to the launcher. Default: ‘’

5.42 parsl.launchers.GnuParallelLauncher

class parsl.launchers.**GnuParallelLauncher**

Worker launcher that wraps the user’s command with the framework to launch multiple command invocations via GNU parallel sshlogin.

This wrapper sets the bash env variable CORES to the number of cores on the machine.

This launcher makes the following assumptions: - GNU parallel is installed and can be located in $PATH - Paswordless SSH login is configured between the controller node and the

target nodes.

- The provider makes available the $PBS_NODEFILE environment variable

__init__()

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

5.43 parsl.launchers.MpiExecLauncher

class parsl.launchers.**MpiExecLauncher**

Worker launcher that wraps the user’s command with the framework to launch multiple command invocations via mpiexec.

This wrapper sets the bash env variable CORES to the number of cores on the machine.

This launcher makes the following assumptions: - mpiexec is installed and can be located in $PATH - The provider makes available the $PBS_NODEFILE environment variable

__init__()

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

5.44 parsl.launchers.JsrunLauncher

class parsl.launchers.**JsrunLauncher**(*overrides=''*)

Worker launcher that wraps the user's command with the Jsrun launch framework to launch multiple cmd invocations in parallel on a single job allocation

__init__(*overrides=''*)

Parameters **overrides** (*str*) – This string will be passed to the JSrun launcher. Default: ''

Methods

__init__([overrides])	**param overrides** This string will be passed to the JSrun launcher. Default: ''

5.45 parsl.monitoring.MonitoringHub

class parsl.monitoring.**MonitoringHub**(*hub_address, hub_port=None, hub_port_range=(55050, 56000), client_address='127.0.0.1', client_port_range=(55000, 56000), workflow_name=None, workflow_version=None, logging_endpoint='sqlite:///monitoring.db', logdir=None, monitoring_debug=False, resource_monitoring_enabled=True, resource_monitoring_interval=30*)

__init__(*hub_address, hub_port=None, hub_port_range=(55050, 56000), client_address='127.0.0.1', client_port_range=(55000, 56000), workflow_name=None, workflow_version=None, logging_endpoint='sqlite:///monitoring.db', logdir=None, monitoring_debug=False, resource_monitoring_enabled=True, resource_monitoring_interval=30*)

Parameters

- **hub_address** (*str*) – The ip address at which the workers will be able to reach the Hub. Default: "127.0.0.1"
- **hub_port** (*int*) – The specific port at which workers will be able to reach the Hub via UDP. Default: None
- **hub_port_range** (*tuple(int, int)*) – The MonitoringHub picks ports at random from the range which will be used by Hub. This is overridden when the hub_port option is set. Defauls: (55050, 56000)
- **client_address** (*str*) – The ip address at which the dfk will be able to reach Hub. Default: "127.0.0.1"
- **client_port_range** (*tuple(int, int)*) – The MonitoringHub picks ports at random from the range which will be used by Hub. Defauls: (55050, 56000)

- **workflow_name** (*str*) – The name for the workflow. Default to the name of the parsl script
- **workflow_version** (*str*) – The version of the workflow. Default to the beginning datetime of the parsl script
- **logging_endpoint** (*str*) – The database connection url for monitoring to log the information. These URLs follow RFC-1738, and can include username, password, hostname, database name. Default: 'sqlite:///monitoring.db'
- **logdir** (*str*) – Parsl log directory paths. Logs and temp files go here. Default: '.'
- **monitoring_debug** (*Bool*) – Enable monitoring debug logging. Default: False
- **resource_monitoring_enabled** (*boolean*) – Set this field to True to enable logging the info of resource usage of each task. Default: True
- **resource_monitoring_interval** (*int*) – The time interval at which the monitoring records the resource usage of each task. Default: 30 seconds

Methods

`__init__`(hub_address[, hub_port, . . .])	**param hub_address** The ip address at which the workers will be able to reach the Hub. Default: "127.0.0.1"
`close`()	
`monitor_wrapper`(f, task_id, . . .)	Internal Wrap the Parsl app with a function that will call the monitor function and point it at the correct pid when the task begins.
`send`(mtype, message)	
`start`(run_id)	

`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.
`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.ControllerError`	Error raise by IPP controller.

Continued on next page

Table 67 – continued from previous page

parsl.executors.errors.ExecutorError	Base class for all exceptions.
parsl.executors.errors.ScalingFailed	Scaling failed due to error in Execution provider.
parsl.executors.errors.InsufficientMPIRanks	Error raised when attempting to launch a MPI worker pool with less than 2 ranks
parsl.executors.errors.DeserializationError	Failure at the Deserialization of results/exceptions from remote workers
parsl.executors.errors.BadMessage	Mangled/Poorly formatted/Unsupported message received
parsl.executors.exceptions.ExecutorException	Base class for all exceptions.
parsl.executors.exceptions.TaskExecException	Task execution raised an error in the remote process.
parsl.dataflow.error.DataFlowException	Base class for all exceptions.
parsl.dataflow.error.ConfigurationError	Raised when the DataFlowKernel receives an invalid configuration.
parsl.dataflow.error.DuplicateTaskError	Raised by the DataFlowKernel when it finds that a job with the same task-id has been launched before.
parsl.dataflow.error.MissingFutError	Raised when a particular future is not found within the dataflowkernel's datastructures.
parsl.dataflow.error.BadCheckpoint	Error raised at the end of app execution due to missing output files.
parsl.dataflow.error.DependencyError	Error raised at the end of app execution due to missing output files.
parsl.launchers.error.BadLauncher	Error raised when a non callable object is provider as Launcher
parsl.providers.error.ExecutionProviderException	Base class for all exceptions Only to be invoked when only a more specific error is not available.
parsl.providers.error.OptionalModuleMissing	Error raised a required module is missing for a optional/extra provider
parsl.providers.error.ChannelRequired	Execution provider requires a channel.
parsl.providers.error.ScaleOutFailed	Generic catch.
parsl.providers.error.SchedulerMissingArgs	Error raised when the template used to compose the submit script to the local resource manager is missing required arguments
parsl.providers.error.ScriptPathError	Error raised when the template used to compose the submit script to the local resource manager is missing required arguments
parsl.channels.errors.ChannelError	Base class for all exceptions
parsl.channels.errors.BadHostKeyException	SSH channel could not be created since server's host keys could not be verified
parsl.channels.errors.BadScriptPath	An error raised during execution of an app.
parsl.channels.errors.BadPermsScriptPath	User does not have permissions to access the script_dir on the remote site
parsl.channels.errors.FileExists	Push or pull of file over channel fails since a file of the name already exists on the destination.
parsl.channels.errors.AuthException	An error raised during execution of an app.
parsl.channels.errors.SSHException	if there was any other error connecting or establishing an SSH session

Continued on next page

Table 67 – continued from previous page

`parsl.channels.errors.FileCopyException`	File copy operation failed
`parsl.executors.high_throughput.errors.WorkerLost`	Exception raised when a worker is lost

5.46 parsl.app.errors.AppBadFormatting

exception `parsl.app.errors.`**`AppBadFormatting`**

An error raised during formatting of a bash function.

5.47 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

5.48 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)

5.49 parsl.app.errors.AppTimeout

exception `parsl.app.errors.`**`AppTimeout`**

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

5.50 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

5.51 parsl.app.errors.BashAppNoReturn

exception `parsl.app.errors.`**`BashAppNoReturn`**(*reason*, *exitcode=-21*, *retries=None*)

Bash app returned no string.

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

5.52 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..)

5.53 parsl.app.errors.InvalidAppTypeError

exception `parsl.app.errors.`**`InvalidAppTypeError`**

An invalid app type was requested from the @App decorator.

5.54 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..)

5.55 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.

5.56 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.

5.57 parsl.executors.errors.ControllerError

exception `parsl.executors.errors.`**`ControllerError`**(*reason*)

Error raise by IPP controller.

5.58 parsl.executors.errors.ExecutorError

exception `parsl.executors.errors.`**`ExecutorError`**(*executor*, *reason*)

Base class for all exceptions.

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

5.59 parsl.executors.errors.ScalingFailed

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

5.60 parsl.executors.errors.InsufficientMPIRanks

exception `parsl.executors.errors.`**`InsufficientMPIRanks`**(*tasks_per_node=None*, *nodes_per_block=None*)
Error raised when attempting to launch a MPI worker pool with less than 2 ranks

5.61 parsl.executors.errors.DeserializationError

exception `parsl.executors.errors.`**`DeserializationError`**(*reason*)
Failure at the Deserialization of results/exceptions from remote workers

5.62 parsl.executors.errors.BadMessage

exception `parsl.executors.errors.`**`BadMessage`**(*reason*)
Mangled/Poorly formatted/Unsupported message received

5.63 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.

5.64 parsl.executors.exceptions.TaskExecException

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

5.65 parsl.dataflow.error.DataFlowException

exception `parsl.dataflow.error.`**`DataFlowException`**
Base class for all exceptions.

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

5.66 parsl.dataflow.error.ConfigurationError

exception `parsl.dataflow.error.`**`ConfigurationError`**

Raised when the DataFlowKernel receives an invalid configuration.

5.67 parsl.dataflow.error.DuplicateTaskError

exception `parsl.dataflow.error.`**`DuplicateTaskError`**

Raised by the DataFlowKernel when it finds that a job with the same task-id has been launched before.

5.68 parsl.dataflow.error.MissingFutError

exception `parsl.dataflow.error.`**`MissingFutError`**

Raised when a particular future is not found within the dataflowkernel's datastructures.

Deprecated.

5.69 parsl.dataflow.error.BadCheckpoint

exception `parsl.dataflow.error.`**`BadCheckpoint`**(*reason*)

Error raised at the end of app execution due to missing output files.

Parameters **`reason`** (–) –

Contains: reason (string) dependent_exceptions

5.70 parsl.dataflow.error.DependencyError

exception `parsl.dataflow.error.`**`DependencyError`**(*dependent_exceptions*, *task_id*, *outputs*)

Error raised at the end of app execution due to missing output files.

Parameters

- **`dependent_exceptions`** (–) – List of exceptions
- **`task_id`** (–) – Identity of the task failed task
- **`outputs ?`** (–) –

Contains: reason (string) dependent_exceptions

5.71 parsl.launchers.error.BadLauncher

exception `parsl.launchers.error.`**`BadLauncher`**(*launcher*, *reason*)

Error raised when a non callable object is provider as Launcher

5.72 parsl.providers.error.ExecutionProviderException

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

5.73 parsl.providers.error.OptionalModuleMissing

exception parsl.providers.error.**OptionalModuleMissing**(*module_names*, *reason*)
Error raised a required module is missing for a optional/extra provider

5.74 parsl.providers.error.ChannelRequired

exception parsl.providers.error.**ChannelRequired**(*provider*, *reason*)
Execution provider requires a channel.

5.75 parsl.providers.error.ScaleOutFailed

exception parsl.providers.error.**ScaleOutFailed**(*provider*, *reason*)
Generic catch. Scale out failed in the submit phase on the provider side

5.76 parsl.providers.error.SchedulerMissingArgs

exception parsl.providers.error.**SchedulerMissingArgs**(*missing_keywords*, *sitename*)
Error raised when the template used to compose the submit script to the local resource manager is missing required arguments

5.77 parsl.providers.error.ScriptPathError

exception parsl.providers.error.**ScriptPathError**(*script_path*, *reason*)
Error raised when the template used to compose the submit script to the local resource manager is missing required arguments

5.78 parsl.channels.errors.ChannelError

exception parsl.channels.errors.**ChannelError**
Base class for all exceptions

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

5.79 parsl.channels.errors.BadHostKeyException

exception `parsl.channels.errors.`**`BadHostKeyException`**(*e*, *hostname*)

SSH channel could not be created since server's host keys could not be verified

Contains: reason(string) e (paramiko exception object) hostname (string)

5.80 parsl.channels.errors.BadScriptPath

exception `parsl.channels.errors.`**`BadScriptPath`**(*e*, *hostname*)

An error raised during execution of an app. What this exception contains depends entirely on context Contains: reason(string) e (paramiko exception object) hostname (string)

5.81 parsl.channels.errors.BadPermsScriptPath

exception `parsl.channels.errors.`**`BadPermsScriptPath`**(*e*, *hostname*)

User does not have permissions to access the script_dir on the remote site

Contains: reason(string) e (paramiko exception object) hostname (string)

5.82 parsl.channels.errors.FileExists

exception `parsl.channels.errors.`**`FileExists`**(*e*, *hostname*, *filename=None*)

Push or pull of file over channel fails since a file of the name already exists on the destination.

Contains: reason(string) e (paramiko exception object) hostname (string)

5.83 parsl.channels.errors.AuthException

exception `parsl.channels.errors.`**`AuthException`**(*e*, *hostname*)

An error raised during execution of an app. What this exception contains depends entirely on context Contains: reason(string) e (paramiko exception object) hostname (string)

5.84 parsl.channels.errors.SSHException

exception `parsl.channels.errors.`**`SSHException`**(*e*, *hostname*)

if there was any other error connecting or establishing an SSH session

Contains: reason(string) e (paramiko exception object) hostname (string)

5.85 parsl.channels.errors.FileCopyException

exception `parsl.channels.errors.`**`FileCopyException`**(*e*, *hostname*)

File copy operation failed

Contains: reason(string) e (paramiko exception object) hostname (string)

5.86 parsl.executors.high_throughput.errors.WorkerLost

exception `parsl.executors.high_throughput.errors.`**`WorkerLost`**(*worker_id*, *hostname*)

Exception raised when a worker is lost

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 0.9.0

Released on October 25th, 2019

Parsl v0.9.0 includes 199 closed issues and pull requests with contributions (code, tests, reviews and reports) from:

Andrew Litteken @AndrewLitteken, Anna Woodard @annawoodard, Ben Clifford @benclifford, Ben Glick @benhg, Daniel S. Katz @danielskatz, Daniel Smith @dgasmith, Engin Arslan @earslan58, Geoffrey Lentner @glentner, John Hover @jhover Kyle Chard @kylechard, TJ Dasso @tjdasso, Ted Summer @macintoshpie, Tom Glanzman @TomGlanzman, Levi Naden @LNaden, Logan Ward @WardLT, Matthew Welborn @mattwelborn, @MatthewBM, Raphael Fialho @rapguit, Yadu Nand Babuji @yadudoc, and Zhuozhao Li @ZhuozhaoLi

New Functionality

- Parsl will no longer do automatic keyword substitution in `@bash_app` in favor of deferring to Python's format method and newer f-strings. For example,

```
# The following example worked until v0.8.0
@bash_app
def cat(inputs=[], outputs=[]):
    return 'cat {inputs[0]} > {outputs[0]}' # <-- Relies on Parsl auto␣
↪formatting the string

# Following are two mechanisms that will work going forward from v0.9.0
@bash_app
def cat(inputs=[], outputs=[]):
    return 'cat {} > {}'.format(inputs[0], outputs[0]) # <-- Use str.
↪format method

@bash_app
def cat(inputs=[], outputs=[]):
    return f'cat {inputs[0]} > {outputs[0]}' # <-- OR use f-strings␣
↪introduced in Python3.6
```

- `@python_app` now takes a `walltime` kwarg to limit the task execution time.
- New file staging API `parsl.data_provider.staging` to support pluggable file staging methods. The methods implemented in 0.8.0 (HTTP(S), FTP and Globus) are still present, along with two new methods which perform HTTP(S) and FTP staging on worker nodes to support non-shared-filesystem executors such as clouds.
- Behaviour change for storage_access parameter. In 0.8.0, this was used to specify Globus staging configuration. In 0.9.0, if this parameter is specified it must specify all desired staging providers. To keep the same staging providers as in 0.8.0, specify:

  ```
  from parsl.data_provider.data_manager import default_staging
  storage_access = default_staging + [GlobusStaging(...)]
  ```

  `GlobusScheme` in 0.8.0 has been renamed GlobusStaging and moved to a new module, parsl.data_provider.globus
- *`WorkQueueExecutor`*: a new executor that integrates functionality from Work Queue is now available.
- New provider to support for Ad-Hoc clusters *`parsl.providers.AdHocProvider`*
- New provider added to support LSF on Summit *`parsl.providers.LSFProvider`*
- Support for CPU and Memory resource hints to providers (github).
- The `logging_level=logging.INFO` in *`MonitoringHub`* is replaced with `monitoring_debug=False`:

  ```
  monitoring=MonitoringHub(
               hub_address=address_by_hostname(),
               hub_port=55055,
               monitoring_debug=False,
               resource_monitoring_interval=10,
  ),
  ```

- Managers now have a worker watchdog thread to report task failures that crash a worker.
- Maximum idletime after which idle blocks can be relinquished can now be configured as follows:

  ```
  config=Config(
               max_idletime=120.0 ,  # float, unit=seconds
               strategy='simple'
  )
  ```

- Several test-suite improvements that have dramatically reduced test duration.
- Several improvements to the Monitoring interface.
- Configurable port on *parsl.channels.SSHChannel*.
- `suppress_failure` now defaults to True.
- *HighThroughputExecutor* is the recommended executor, and *IPyParallelExecutor* is deprecated.
- *HighThroughputExecutor* will expose worker information via environment variables: `PARSL_WORKER_RANK` and `PARSL_WORKER_COUNT`

Bug Fixes

- ZMQError: Operation cannot be accomplished in current state bug issue#1146
- Fix event loop error with monitoring enabled issue#532
- Tasks per app graph appears as a sawtooth, not as rectangles issue#1032.
- Globus status processing failure issue#1317.
- Sporadic globus staging error issue#1170.
- RepresentationMixin breaks on classes with no default parameters issue#1124.
- File `localpath` staging conflict issue#1197.
- Fix IndexError when using CondorProvider with strategy enabled issue#1298.
- Improper dependency error handling causes hang issue#1285.
- Memoization/checkpointing fixes for bash apps issue#1269.
- CPU User Time plot is strangely cumulative issue#1033.
- Issue requesting resources on non-exclusive nodes issue#1246.
- parsl + htex + slurm hangs if slurm command times out, without making further progress issue#1241.
- Fix strategy overallocations issue#704.
- max_blocks not respected in SlurmProvider issue#868.
- globus staging does not work with a non-default `workdir` issue#784.
- Cumulative CPU time loses time when subprocesses end issue#1108.
- Interchange KeyError due to too many heartbeat missed issue#1128.

6.2.2 Parsl 0.8.0

Released on June 13th, 2019

Parsl v0.8.0 includes 58 closed issues and pull requests with contributions (code, tests, reviews and reports)

from: Andrew Litteken @AndrewLitteken, Anna Woodard @annawoodard, Antonio Villarreal @villarrealas, Ben Clifford @benc, Daniel S. Katz @danielskatz, Eric Tatara @etatara, Juan David Garrido @garri1105, Kyle Chard @@kylechard, Lindsey Gray @lgray, Tim Armstrong @timarmstrong, Tom Glanzman @TomGlanzman, Yadu Nand Babuji @yadudoc, and Zhuozhao Li @ZhuozhaoLi

New Functionality

- Monitoring is now integrated into parsl as default functionality.
- `parsl.AUTO_LOGNAME`: Support for a special `AUTO_LOGNAME` option to auto generate `stdout` and `stderr` file paths.
- `parsl.Files` no longer behave as strings. This means that operations in apps that treated `parsl.Files` as strings will break. For example the following snippet will have to be updated:

```
# Old style: " ".join(inputs) is legal since inputs will behave like a list of␣
↪strings
@bash_app
def concat(inputs=[], outputs=[], stdout="stdout.txt", stderr='stderr.txt'):
    return "cat {0} > {1}".format(" ".join(inputs), outputs[0])

# New style:
@bash_app
def concat(inputs=[], outputs=[], stdout="stdout.txt", stderr='stderr.txt'):
    return "cat {0} > {1}".format(" ".join(list(map(str,inputs))), outputs[0])
```

- Cleaner user app file log management.
- Updated configurations using *`HighThroughputExecutor`* in the configuration section of the userguide.
- Support for OAuth based SSH with *`OAuthSSHChannel`*.

Bug Fixes

- Monitoring resource usage bug issue#975
- Bash apps fail due to missing dir paths issue#1001
- Viz server explicit binding fix issue#1023
- Fix sqlalchemy version warning issue#997
- All workflows are called typeguard issue#973
- Fix `ModuleNotFoundError:  No module named 'monitoring` issue#971
- Fix sqlite3 integrity error issue#920
- HTEX interchange check python version mismatch to the micro level issue#857
- Clarify warning message when a manager goes missing issue#698
- Apps without a specified DFK should use the global DFK in scope at call time, not at other times. issue#697

6.2.3 Parsl 0.7.2

Released on Mar 14th, 2019

New Functionality

- `Monitoring`: Support for reporting monitoring data to a local sqlite database is now available.
- Parsl is switching to an opt-in model for anonymous usage tracking. Read more here: *Usage statistics collection*.
- *`bash_app`* now supports specification of write modes for `stdout` and `stderr`.

- Persistent volume support added to `Kubernetes` provider.
- Scaling recommendations from study on Bluewaters is now available in the userguide.

6.2.4 Parsl 0.7.1

Released on Jan 18th, 2019

New Functionality

- `LowLatencyExecutor`: a new executor designed to address use-cases with tight latency requirements such as model serving (Machine Learning), function serving and interactive analyses is now available.
- **New options in `HighThroughputExecutor`:**
  - `suppress_failure`: Enable suppression of worker rejoin errors.
  - `max_workers`: Limit workers spawned by manager
- Late binding of DFK, allows apps to pick DFK dynamically at call time. This functionality adds safety to cases where a new config is loaded and a new DFK is created.

Bug fixes

- A critical bug in `HighThroughputExecutor` that led to debug logs overflowing channels and terminating blocks of resource is fixed issue#738

6.2.5 Parsl 0.7.0

Released on Dec 20st, 2018

Parsl v0.7.0 includes 110 closed issues with contributions (code, tests, reviews and reports) from: Alex Hays @ahayschi, Anna Woodard @annawoodard, Ben Clifford @benc, Connor Pigg @ConnorPigg, David Heise @daheise, Daniel S. Katz @danielskatz, Dominic Fitzgerald @djf604, Francois Lanusse @EiffL, Juan David Garrido @garri1105, Gordon Watts @gordonwatts, Justin Wozniak @jmjwozniak, Joseph Moon @jmoon1506, Kenyi Hurtado @khurtado, Kyle Chard @kylechard, Lukasz Lacinski @lukaszlacinski, Ravi Madduri @madduri, Marco Govoni @mgovoni-devel, Reid McIlroy-Young @reidmcy, Ryan Chard @ryanchard, @sdustrud, Yadu Nand Babuji @yadudoc, and Zhuozhao Li @ZhuozhaoLi

New functionality

- `HighThroughputExecutor`: a new executor intended to replace the `IPyParallelExecutor` is now available. This new executor addresses several limitations of `IPyParallelExecutor` such as:
  - Scale beyond the ~300 worker limitation of IPP.
  - Multi-processing manager supports execution on all cores of a single node.
  - Improved worker side reporting of version, system and status info.
  - Supports failure detection and cleaner manager shutdown.

  Here's a sample configuration for using this executor locally:

```
from parsl.providers import LocalProvider
from parsl.channels import LocalChannel

from parsl.config import Config
from parsl.executors import HighThroughputExecutor

config = Config(
    executors=[
        HighThroughputExecutor(
            label="htex_local",
            cores_per_worker=1,
            provider=LocalProvider(
                channel=LocalChannel(),
                init_blocks=1,
                max_blocks=1,
            ),
        )
    ],
)
```

More information on configuring is available in the *Configuration* section.

- `ExtremeScaleExecutor` a new executor targeting supercomputer scale (>1000 nodes) workflows is now available.

  Here's a sample configuration for using this executor locally:

```
from parsl.providers import LocalProvider
from parsl.channels import LocalChannel
from parsl.launchers import SimpleLauncher

from parsl.config import Config
from parsl.executors import ExtremeScaleExecutor

config = Config(
    executors=[
        ExtremeScaleExecutor(
            label="extreme_local",
            ranks_per_node=4,
            provider=LocalProvider(
                channel=LocalChannel(),
                init_blocks=0,
                max_blocks=1,
                launcher=SimpleLauncher(),
            )
        )
    ],
    strategy=None,
)
```

  More information on configuring is available in the *Configuration* section.

- The libsubmit repository has been merged with Parsl to reduce overheads on maintenance with respect to documentation, testing, and release synchronization. Since the merge, the API has undergone several updates to support the growing collection of executors, and as a result Parsl 0.7.0+ will not be backwards compatible with the standalone libsubmit repos. The major components of libsubmit are now available through Parsl, and require the following changes to import lines to migrate scripts to 0.7.0:

  – `from libsubmit.providers import <ProviderName>` is now `from parsl.`

providers import <ProviderName>

  - from libsubmit.channels import <ChannelName> is now from parsl.channels import <ChannelName>
  - from libsubmit.launchers import <LauncherName> is now from parsl.launchers import <LauncherName>

Warning: This is a breaking change from Parsl v0.6.0

- To support resource-based requests for workers and to maintain uniformity across interfaces, `tasks_per_node` is no longer a **provider** option. Instead, the notion of `tasks_per_node` is defined via executor specific options, for eg:
  - *`IPyParallelExecutor`* provides `workers_per_node`
  - *`HighThroughputExecutor`* provides `cores_per_worker` to allow for worker launches to be determined based on the number of cores on the compute node.
  - *`ExtremeScaleExecutor`* uses `ranks_per_node` to specify the ranks to launch per node.

Warning: This is a breaking change from Parsl v0.6.0

- **Major upgrades to the monitoring infrastructure.**
  - Monitoring information can now be written to a SQLite database, created on the fly by Parsl
  - Web-based monitoring to track workflow progress
- Determining the correct IP address/interface given network firewall rules is often a nuisance. To simplify this, three new methods are now supported:
  - `parsl.addresses.address_by_route`
  - `parsl.addresses.address_by_query`
  - `parsl.addresses.address_by_hostname`
- *`AprunLauncher`* now supports `overrides` option that allows arbitrary strings to be added to the aprun launcher call.
- *`DataFlowKernel`* has a new method `wait_for_current_tasks()`
- *`DataFlowKernel`* now uses per-task locks and an improved mechanism to handle task completions improving performance for workflows with large number of tasks.

Bug fixes (highlights)

- Ctlr+C should cause fast DFK cleanup issue#641
- Fix to avoid padding in `wtime_to_minutes()` issue#522
- Updates to block semantics issue#557
- Updates `public_ip` to `address` for clarity issue#557
- Improvements to launcher docs issue#424
- Fixes for inconsistencies between stream_logger and file_logger issue#629

- Fixes to DFK discarding some un-executed tasks at end of workflow issue#222
- Implement per-task locks to avoid deadlocks issue#591
- Fixes to internal consistency errors issue#604
- Removed unnecessary provider labels issue#440
- Fixes to `TorqueProvider` to work on NSCC issue#489
- Several fixes and updates to monitoring subsystem issue#471
- DataManager calls wrong DFK issue#412
- Config isn't reloading properly in notebooks issue#549
- Cobalt provider `partition` should be `queue` issue#353
- bash AppFailure exceptions contain useful but un-displayed information issue#384
- Do not CD to engine_dir issue#543
- Parsl install fails without kubernetes config file issue#527
- Fix import error issue#533
- Change Local Database Strategy from Many Writers to a Single Writer issue#472
- All run-related working files should go in the rundir unless otherwise configured issue#457
- Fix concurrency issue with many engines accessing the same IPP config issue#469
- Ensure we are not caching failed tasks issue#368
- File staging of unknown schemes fails silently issue#382
- Inform user checkpointed results are being used issue#494
- Fix IPP + python 3.5 failure issue#490
- File creation fails if no executor has been loaded issue#482
- Make sure tasks in `dep_fail` state are retried issue#473
- Hard requirement for CMRESHandler issue#422
- Log error Globus events to stderr issue#436
- Take 'slots' out of logging issue#411
- Remove redundant logging issue#267
- Zombie ipcontroller processes - Process cleanup in case of interruption issue#460
- IPyparallel failure when submitting several apps in parallel threads issue#451
- `SlurmProvider` + `SingleNodeLauncher` starts all engines on a single core issue#454
- IPP `engine_dir` has no effect if indicated dir does not exist issue#446
- Clarify AppBadFormatting error issue#433
- confusing error message with simple configs issue#379
- Error due to missing kubernetes config file issue#432
- `parsl.configs` and `parsl.tests.configs` missing init files issue#409
- Error when Python versions differ issue#62
- Fixing ManagerLost error in HTEX/EXEX issue#577

- Write all debug logs to rundir by default in HTEX/EXEX issue#574
- Write one log per HTEX worker issue#572
- Fixing ManagerLost error in HTEX/EXEX issue#577

6.2.6 Parsl 0.6.1

Released on July 23rd, 2018.

This point release contains fixes for issue#409

6.2.7 Parsl 0.6.0

Released July 23rd, 2018.

New functionality

- Switch to class based configuration issue#133

  Here's a the config for using threads for local execution

```
from parsl.config import Config
from parsl.executors.threads import ThreadPoolExecutor

config = Config(executors=[ThreadPoolExecutor()])
```

  Here's a more complex config that uses SSH to run on a Slurm based cluster

```
from libsubmit.channels import SSHChannel
from libsubmit.providers import SlurmProvider

from parsl.config import Config
from parsl.executors.ipp import IPyParallelExecutor
from parsl.executors.ipp_controller import Controller

config = Config(
    executors=[
        IPyParallelExecutor(
            provider=SlurmProvider(
                'westmere',
                channel=SSHChannel(
                    hostname='swift.rcc.uchicago.edu',
                    username=<USERNAME>,
                    script_dir=<SCRIPTDIR>
                ),
                init_blocks=1,
                min_blocks=1,
                max_blocks=2,
                nodes_per_block=1,
                tasks_per_node=4,
                parallelism=0.5,
                overrides=<SPECIFY_INSTRUCTIONS_TO_LOAD_PYTHON3>
            ),
            label='midway_ipp',
            controller=Controller(public_ip=<PUBLIC_IP>),
```

(continues on next page)

(continued from previous page)

```
        )
    ]
)
```

- Implicit Data Staging issue#281
- Support for application profiling issue#5
- Real-time usage tracking via external systems issue#248, issue#251
- Several fixes and upgrades to tests and testing infrastructure issue#157, issue#159, issue#128, issue#192, issue#196
- Better state reporting in logs issue#242
- Hide DFK issue#50
  - Instead of passing a config dictionary to the DataFlowKernel, now you can call `parsl.load(Config)`
  - Instead of having to specify the `dfk` at the time of `App` declaration, the DFK is a singleton loaded at call time :

```
import parsl
from parsl.tests.configs.local_ipp import config
parsl.load(config)

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

fut = double(5)
fut.result()
```

- Support for better reporting of remote side exceptions issue#110

Bug Fixes

- Making naming conventions consistent issue#109
- Globus staging returns unclear error bug issue#178
- Duplicate log-lines when using IPP issue#204
- Usage tracking with certain missing network causes 20s startup delay. issue#220
- `task_exit` checkpointing repeatedly truncates checkpoint file during run bug issue#230
- Checkpoints will not reload from a run that was Ctrl-C'ed issue#232
- Race condition in task checkpointing issue#234
- Failures not to be checkpointed issue#239
- Naming inconsitencies with `maxThreads`, `max_threads`, `max_workers` are now resolved issue#303
- Fatal not a git repository alerts issue#326
- Default `kwargs` in bash apps unavailable at command-line string format time issue#349
- Fix launcher class inconsistencies issue#360
- **Several fixes to AWS provider issue#362**

  - Fixes faulty status updates
  - Faulty termination of instance at cleanup, leaving zombie nodes.

6.2.8 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
- 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

  Parsl 0.5.2

Released. June 21st, 2018. This is an emergency release addressing issue#347

Bug Fixes

- Parsl version conflict with libsubmit 0.4.1 issue#347

6.2.9 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},
              "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

6.2.10 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.

6.2.11 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*, *App caching*

- 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

6.2.12 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.

6.2.13 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

6.2.14 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

6.2.15 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 Libsubmit Changelog

As of Parsl 0.7.0 the libsubmit repository has been merged into Parsl.

6.3.1 Libsubmit 0.4.1

Released. June 18th, 2018. This release folds in massive contributions from @annawoodard.

New functionality

- Several code cleanups, doc improvements, and consistent naming
- All providers have the initialization and actual start of resources decoupled.

6.3.2 Libsubmit 0.4.0

Released. May 15th, 2018. This release folds in contributions from @ahayschi, @annawoodard, @yadudoc

New functionality

- Several enhancements and fixes to the AWS cloud provider (#44, #45, #50)
- Added support for python3.4

Bug Fixes

- Condor jobs left in queue with X state at end of completion issue#26
- Worker launches on Cori seem to fail from broken ENV issue#27
- EC2 provider throwing an exception at initial run issue#46

Design and Rationale

6.4 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.

6.4.1 Parallel Evaluation

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

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

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

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

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

foreach v in array {
   trace(v)
}
```

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

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

```
futures = {};

for i in range(0,10):
```

(continues on next page)

(continued from previous page)

```
    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.

6.4.2 Mappers

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

There are two choices here :

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

In Swift, the user defines file mappings like this :

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

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

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

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

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

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

6.4.3 Remote-Execution

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

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

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

- [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.

6.4.4 Availability of Python3.5 on target resources

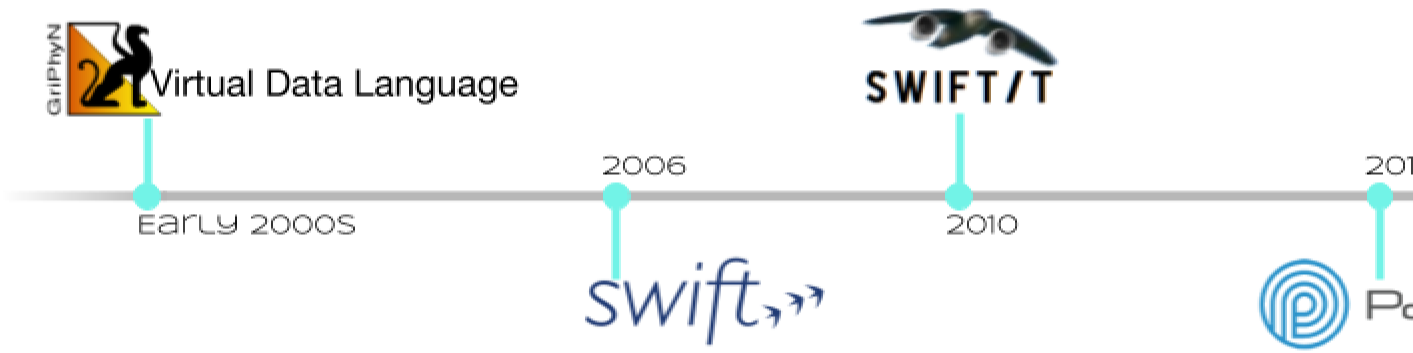
The availability of Python3.5 on compute resources, especially one's on which the user does not have admin privileges could be a concern. This was raised by Lincoln from the OSG Team. Here's a small table of our initial target systems as of Mar 3rd, 2017 :

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

Under construction.

6.5 Roadmap

Before diving into the roadmap, a quick retrospective look at the evolution of workflow solutions that came before Parsl from the workflows group at UChicago and Argonne National Laboratory.



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 v0.9.0 and features in preparation are documented via Github issues and milestones.

6.5.1 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).**
- **Class-based config definition (v0.6.0)**
- **Singleton config, and separate DFK from app definitions (v0.6.0)**
- Class-based app definition

6.5.2 Data management

- **File abstraction to support representation of local and remote files.**
- **Support for a variety of common data access protocols (e.g., FTP, HTTP, Globus) (v0.6.0)**.
- **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) and worker node (v0.6.0)**.
- Support for creation of a sandbox and execution within the sandbox.
- Multi-site support including transparent movement between sites.
- **Support for systems without a shared file system (point-to-point staging). (Partial support in v0.9.0)**
- Support for data caching at multiple levels and across sites.

6.5.3 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. (v0.5.0)**

- Add data staging to task state transition model.
- **More efficient algorithms for managing dependency resolution. (v0.7.0)**
- Scheduling and allocation algorithms that determine job placement based on job and data requirements (including deadlines) as well as site capabilities.
- **Directing jobs to a specific set of sites.(v0.4.0)**
- **Logic to manage (provision, resize) execution resource block based on job requirements, and running multiple tasks per resource block (v0.4.0).**
- **Retry logic to support recovery and fault tolerance**
- **Workflow level checkpointing and restart (v0.4.0)**
- **Transition away from IPP to in-house executors (HighThroughputExecutor and ExtremeScaleExecutor v0.7.0)**

6.5.4 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**
    - **HTCondor**
    - **Cobalt**
    - **GridEngine**
    - **PBS/Torque**
    - **AWS**
    - **GoogleCloud**
    - **Azure**
    - **Nova/OpenStack/Jetstream (partial support)**
    - **Kubernetes (v0.6.0)**
- **Support for launcher mechanisms**
    - **srun**
    - **aprun (Complete support 0.6.0)**
    - **Various MPI launch mechanisms (Mpiexec, mpirun..)**
- **Support for remote execution using SSH (from v0.3.0)and OAuth-based authentication (from v0.9.0)**
- **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
- **IPP workers to support multiple threads of execution per node. (v0.7.0 adds support via replacement executors)**
- Smarter serialization with caching frequently used objects.

- **Support for user-defined containers as Parsl apps and orchestration of workflows comprised of containers (v0.5.0)**
  - **Docker (locally)**
  - Shifter (NERSC, Blue Waters)
  - Singularity (ALCF)

6.5.5 Visualization, debugging, fault tolerance

- **Support for exception handling**.
- **Interface for accessing real-time state (v0.6.0)**.
- **Visualization library that enables users to introspect graph, task, and data dependencies, as well as observe state of executed/executing tasks (from v0.9.0)**
- 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**
- **Support for saving monitoring information to local DB (sqlite) and remote DB (elasticsearch) (v0.6.0 and v0.7.0)**

6.5.6 Authentication and authorization

- **Seamless authentication using OAuth-based methods within Parsl scripts (e.g., native app grants) (v0.6.0)**
- Support for arbitrary identity providers and pass through to execution resources
- Support for transparent/scoped access to external services **(e.g., Globus transfer) (v0.6.0)**

6.5.7 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)**

6.5.8 Documentation / Tutorials:

- **Documentation about Parsl and its features**
- **Documentation about supported sites (v0.6.0)**
- **Self-guided Jupyter notebook tutorials on Parsl features**
- **Hands-on tutorial suitable for webinars and meetings**

6.6 Developer Guide

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

6.6.1 Importing

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

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

6.6.2 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.

6.6.3 Constants

AUTO_LOGNAME Special value that indicates Parsl should construct a filename for logging.

`parsl.`**`set_stream_logger`**(*name: str = 'parsl', level: int = 10, format_string: Optional[str] = None*)

Add a stream log handler.

Parameters

- **`name`** (–) – Set the logger name.
- **`level`** (–) – Set to logging.DEBUG by default.
- **`format_string`** (–) – Set to None by default.

Returns

- None

`parsl.`**`set_file_logger`**(*filename: str, name: str = 'parsl', level: int = 10, format_string: Optional[str] = None*)

Add a stream log handler.

Parameters

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

Returns

- None

6.6.4 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*, *data_flow_kernel=None*, *walltime=60*, *executors='all'*, *cache=False*)

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.`**`PythonApp`**(*func*, *data_flow_kernel=None*, *walltime=60*, *cache=False*, *executors='all'*)

Extends AppBase to cover the Python App.

BashApp

Concrete subclass of AppBase that implements the Bash App functionality.

class `parsl.app.bash.`**`BashApp`**(*func*, *data_flow_kernel=None*, *walltime=60*, *cache=False*, *executors='all'*)

6.6.5 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`**(*task_def*)

An AppFuture wraps a sequence of Futures which may fail and be retried.

The AppFuture will wait for the DFK to provide a result from an appropriate parent future, through `parent_callback`. It will set its result to the result of that parent future, if that parent future completes without an exception. This result setting should cause .result(), .exception() and done callbacks to fire as expected.

The AppFuture will not set its result to the result of the parent future, if that parent future completes with an exception, and if that parent future has retries left. In that case, no result(), exception() or done callbacks should report a result.

The AppFuture will set its result to the result of the parent future, if that parent future completes with an exception and if that parent future has no retries left, or if it has no retry field. .result(), .exception() and done callbacks should give a result as expected when a Future has a result set

The parent future may return a RemoteExceptionWrapper as a result and AppFuture will treat this an an exception for the above retry and result handling behaviour.

`__init__`(*task_def*)
Initialize the AppFuture.

Args:

KWargs:

- **task_def** [The DFK task definition dictionary for the task represented] by this future.

`__repr__`()
Return repr(self).

`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.

`parent_callback`(*executor_fu*)
Callback from a parent future to update the AppFuture.

Used internally by AppFuture, and should not be called by code using AppFuture.

Parameters **`executor_fu`** (–) – Future returned by the executor along with callback.

Returns

- None

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

DataFutures

class `parsl.app.futures.`**`DataFuture`**(*fut*, *file_obj*, *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*, *tid=None*)
Construct the DataFuture object.

If the file_obj is a string convert to a File.

Parameters

- **`fut`** (–) – AppFuture that this DataFuture will track
- **`file_obj`** (–) – Something representing file(s)

Kwargs:

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

`__repr__`()
Return repr(self).

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.

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.

Updates the future with the result (the File object) or the parent future's exception.

Parameters **parent_fu** (–) – Future returned by the executor along with callback

Returns

- None

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

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

6.6.6 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**
An error raised during formatting of a bash function.

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.

6.6.7 DataFlowKernel

class parsl.dataflow.dflow.**DataFlowKernel**(*config=Config(app_cache=True, checkpoint_files=None, checkpoint_mode=None, checkpoint_period=None, data_management_max_threads=10, executors=[ThreadPoolExecutor(label='threads', managed=True, max_threads=2, storage_access=None, thread_name_prefix='', working_dir=None)], initialize_logging=True, lazy_errors=True, max_idletime=120.0, monitoring=None, retries=0, run_dir='runinfo', strategy='simple', usage_tracking=False)*)
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 is a simplified diagram of what happens internally:

```
 User                |           DFK          |     Executor
----------------------------------------------------------
                     |                        |
      Task-------+> +Submit                   |
    App_Fu<------+--|                         |
                     |  Dependencies met      |
                     |         task-------+--> +Submit
                     |        Ex_Fu<------+----|
```

__init__(*config=Config(app_cache=True, checkpoint_files=None, checkpoint_mode=None, checkpoint_period=None, data_management_max_threads=10, executors=[ThreadPoolExecutor(label='threads', managed=True, max_threads=2, storage_access=None, thread_name_prefix='', working_dir=None)], initialize_logging=True, lazy_errors=True, max_idletime=120.0, monitoring=None, retries=0, run_dir='runinfo', strategy='simple', usage_tracking=False)*)
Initialize the DataFlowKernel.

Parameters config (Config) – A specification of all configuration options. For more details see the :class:~'parsl.config.Config' documentation.

`__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 releasing all resources explicitly.

If the executors are managed by the DFK, then we call scale_in on each of the executors and call executor.shutdown. Otherwise, executor cleanup is left to the user.

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

Returns

- config (dict)

`handle_app_update`(*task_id*, *future*, *memo_cbk=False*)
This function is called as a callback when an AppFuture is in its final state.

It will trigger post-app processing such as checkpointing.

Parameters

- **`task_id`** (*`string`*) – Task id
- **`future`** (*`Future`*) – The relevant app future (which should be consistent with the task structure 'app_fu' entry

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

`handle_exec_update`(*task_id*, *future*)
This function is called only as a callback from an execution attempt reaching a final state (either successfully or failing).

It will launch retries if necessary, and update the task structure.

Parameters

- **`task_id`** (*`string`*) – Task id which is a uuid string
- **`future`** (*`Future`*) – The future object corresponding to the task which
- **`this callback`** (*`makes`*) –

launch_if_ready(*task_id*)

launch_if_ready will launch the specified task, if it is ready to run (for example, without dependencies, and in pending state).

This should be called by any piece of the DataFlowKernel that thinks a task may have become ready to run.

It is not an error to call launch_if_ready on a task that is not ready to run - launch_if_ready will not incorrectly launch that task.

launch_if_ready is thread safe, so may be called from any thread or callback.

launch_task(*task_id*, *executable*, **args*, ***kwargs*)

Handle the actual submission of the task to the executor layer.

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

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

Parameters

- **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

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.

Parameters

- **task_id** (*uuid str*) – Task id
- **func** (*Function*) – App function
- **args** (*List*) – Positional args to app function
- **kwargs** (*Dict*) – Kwargs to app function

Returns partial function evaluated with all dependencies in args, kwargs and kwargs['inputs'] evaluated.

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

If the app task has the executors attributes not set (default=='all') the task will be launched on a randomly selected executor from the list of executors. If the app task specifies a particular set of executors, it will be targeted at the specified executors.

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

Parameters

- **func** (–) – A function object
- ***args** (–) – Args to the function

KWargs :

- **executors (list or string)** [List of executors 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,]

wait_for_current_tasks ()
Waits for all tasks in the task list to be completed, by waiting for their AppFuture to be completed. This method will not necessarily wait for any tasks added after cleanup has started (such as data stageout?)

6.6.8 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.

In addition to the listed methods, a ParslExecutor instance must always have a member field:

label: str - a human readable label for the executor, unique with respect to other executors.

An executor may optionally expose:

storage_access: List[parsl.data_provider.staging.Staging] - a list of staging providers that will be used for file staging. In the absence of this attribute, or if this attribute is `None`, then a default value of `parsl.data_provider.staging.default_staging` will be used by the staging code.

Typechecker note: Ideally storage_access would be declared on executor __init__ methods as List[Staging] - however, lists are by default invariant, not co-variant, and it looks like @typeguard cannot be persuaded otherwise. So if you're implementing an executor and want to @typeguard the constructor, you'll have to use List[Any] here.

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

`scale_in` (*blocks: int*) → None
Scale in method.

Cause the executor to reduce the number of blocks by count.

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` (*blocks: int*) → None
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` (*func: Callable*, **args*, ***kwargs*) → concurrent.futures._base.Future
Submit.

The value returned must be a Future, with the further requirements that it must be possible to assign a retries_left member slot to that object.

ThreadPoolExecutor

class `parsl.executors.threads.`**`ThreadPoolExecutor`**(*label: str = 'threads'*, *max_threads: int = 2*, *thread_name_prefix: str = ''*, *storage_access: List[Any] = None*, *working_dir: Optional[str] = None*, *managed: bool = True*)

A thread-based executor.

Parameters

- **`max_threads`** (`int`) – Number of threads. Default is 2.
- **`thread_name_prefix`** (`string`) – Thread name prefix (only supported in python v3.6+).
- **`storage_access`** (list of `Staging`) – Specifications for accessing data this executor remotely.
- **`managed`** (`bool`) – If True, parsl will control dynamic scaling of this executor, and be responsible. Otherwise, this is managed by the user.

`__init__`(*label: str = 'threads', max_threads: int = 2, thread_name_prefix: str = '', storage_access: List[Any] = None, working_dir: Optional[str] = None, managed: bool = True*)
Initialize self. See help(type(self)) for accurate signature.

`scale_in`(*blocks*)
Scale in the number of active blocks by specified amount.

This method is not implemented 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

`start`()
Start the executor.

Any spin-up operations (for example: starting thread pools) should be performed here.

`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:

IPyParallelExecutor

Warning: Deprecated as of `v0.9.0`

class `parsl.executors.ipp.`**`IPyParallelExecutor`**(*provider=LocalProvider(channel=LocalChannel(envs={}, script_dir=None, userhome='/home/docs/checkouts/readthedocs.org/user_builds/parsl/c), cmd_timeout=30, init_blocks=4, launcher=SingleNodeLauncher(), max_blocks=10, min_blocks=0, move_files=None, nodes_per_block=1, parallelism=1, walltime='00:15:00', worker_init=''), label='ipp', working_dir=None, controller=Controller(interfaces=None, ipython_dir='~/.ipython', log=True, mode='auto', port=None, port_range=None, profile='default', public_ip=None, reuse=False), container_image=None, engine_dir=None, storage_access=None, engine_debug_level=None, workers_per_node=1, managed=True*)

The IPython Parallel executor.

This executor uses IPythonParallel's pilot execution system to manage multiple processes running locally or remotely.

Parameters

- **`provider`** (*`ExecutionProvider`*) – Provider to access computation resources. Can be one of `EC2Provider`, `Cobalt`, `Condor`, `GoogleCloud`, `GridEngine`, `Jetstream`, `Local`, `GridEngine`, `Slurm`, or `Torque`.
- **`label`** (*`str`*) – Label for this executor instance.
- **`controller`** (*`Controller`*) – Which Controller instance to use. Default is *`Controller()`*.
- **`workers_per_node`** (*`int`*) – Number of workers to be launched per node. Default=1
- **`container_image`** (*`str`*) – Launch tasks in a container using this docker image. If set to None, no container is used. Default is None.
- **`engine_dir`** (*`str`*) – Directory where engine logs and configuration files will be stored.
- **`working_dir`** (*`str`*) – Directory where input data should be staged to.
- **`storage_access`** (list of `Staging`) – Specifications for accessing data this executor remotely.
- **`managed`** (*`bool`*) – If True, parsl will control dynamic scaling of this executor, and be responsible. Otherwise, this is managed by the user.
- **`engine_debug_level`** (*`int | str`*) – Sets engine logging to specified debug level. Choices: (0, 10, 20, 30, 40, 50, 'DEBUG', 'INFO', 'WARN', 'ERROR', 'CRITICAL')

:param .. note::: Some deficiencies with this executor are:

1. Ipengines 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__`(*provider=LocalProvider(channel=LocalChannel(envs={}, script_dir=None, userhome='/home/docs/checkouts/readthedocs.org/user_builds/parsl/checkouts/0.9.0/docs'), cmd_timeout=30, init_blocks=4, launcher=SingleNodeLauncher(), max_blocks=10, min_blocks=0, move_files=None, nodes_per_block=1, parallelism=1, walltime='00:15:00', worker_init=''), label='ipp', working_dir=None, controller=Controller(interfaces=None, ipython_dir='~/.ipython', log=True, mode='auto', port=None, port_range=None, profile='default', public_ip=None, reuse=False), container_image=None, engine_dir=None, storage_access=None, engine_debug_level=None, workers_per_node=1, managed=True*)

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

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

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

Parameters

- **`filepath`** – Path to the engine file
- **`engine_dir`** – CWD for the engines

`scale_in`(*blocks*)

Scale in the number of active blocks by the specified number.

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

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

Parameters `blocks` – int Number of blocks to be provisioned.

`scaling_enabled`
Specify if scaling is enabled.

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

`start`()
Start the executor.

Any spin-up operations (for example: starting thread pools) should be performed here.

`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

HighThroughputExecutor

class parsl.executors.**HighThroughputExecutor**(*label: str = 'HighThroughputExecutor', provider: parsl.providers.provider_base.ExecutionProvider = LocalProvider(channel=LocalChannel(envs={}, script_dir=None, userhome='/home/docs/checkouts/readthedocs.org/user_builds/parsl/ch), cmd_timeout=30, init_blocks=4, launcher=SingleNodeLauncher(), max_blocks=10, min_blocks=0, move_files=None, nodes_per_block=1, parallelism=1, walltime='00:15:00', worker_init=''), launch_cmd: Optional[str] = None, address: str = '127.0.0.1', worker_ports: Optional[Tuple[int, int]] = None, worker_port_range: Optional[Tuple[int, int]] = (54000, 55000), interchange_port_range: Optional[Tuple[int, int]] = (55000, 56000), storage_access: Optional[List[parsl.data_provider.staging.Staging]] = None, working_dir: Optional[str] = None, worker_debug: bool = False, cores_per_worker: float = 1.0, mem_per_worker: Optional[float] = None, max_workers: Union[int, float] = inf, prefetch_capacity: int = 0, heartbeat_threshold: int = 120, heartbeat_period: int = 30, poll_period: int = 10, suppress_failure: bool = True, managed: bool = True, worker_logdir_root: Optional[str] = None*)

Executor designed for cluster-scale

The HighThroughputExecutor system has the following components:

1. The HighThroughputExecutor instance which is run as part of the Parsl script.
2. The Interchange which is acts as a load-balancing proxy between workers and Parsl
3. The multiprocessing based worker pool which coordinates task execution over several cores on a node.
4. ZeroMQ pipes connect the HighThroughputExecutor, Interchange and the process_worker_pool

Here is a diagram

```
             |  Data   |  Executor    |  Interchange  | External Process(es)
             |  Flow   |              |               |
        Task | Kernel  |              |               |
      +----->|-------->|------------->|->outgoing_q---|-> process_worker_pool
      |      |         |              | batching      |    |         |
Parsl<---Fut-|         |              | load-balancing|  result   exception
          ^  |         |              | watchdogs     |    |         |
          |  |         |   Q_mngmnt   |               |    V         V
          |  |         |    Thread<--|-incoming_q<---|--- +---------+
```

(continues on next page)

(continued from previous page)

```
          |  |          |       |       |               |
          |  |          |       |       |               |
          +----update_fut-----+
```

Each of the workers in each process_worker_pool has access to its local rank through an environmental variable, `PARSL_WORKER_RANK`. The local rank is unique for each process and is an integer in the range from 0 to the number of workers per in the pool minus 1. The workers also have access to the ID of the worker pool as `PARSL_WORKER_POOL_ID` and the size of the worker pool as `PARSL_WORKER_COUNT`.

Parameters

- **provider** (*ExecutionProvider*) –

  Provider to access computation resources. Can be one of EC2Provider, `Cobalt`, `Condor`, `GoogleCloud`, `GridEngine`, `Jetstream`, `Local`, `GridEngine`, `Slurm`, or `Torque`.
- **label** (*str*) – Label for this executor instance.
- **launch_cmd** (*str*) – Command line string to launch the process_worker_pool from the provider. The command line string will be formatted with appropriate values for the following values (debug, task_url, result_url, cores_per_worker, nodes_per_block, heartbeat_period ,heartbeat_threshold, logdir). For example: launch_cmd="process_worker_pool.py {debug} -c {cores_per_worker} –task_url={task_url} –result_url={result_url}"
- **address** (*string*) – An address to connect to the main Parsl process which is reachable from the network in which workers will be running. This can be either a hostname as returned by `hostname` or an IP address. Most login nodes on clusters have several network interfaces available, only some of which can be reached from the compute nodes. Some trial and error might be necessary to identify what addresses are reachable from compute nodes.
- **worker_ports** (*(int, int)*) – Specify the ports to be used by workers to connect to Parsl. If this option is specified, worker_port_range will not be honored.
- **worker_port_range** (*(int, int)*) – Worker ports will be chosen between the two integers provided.
- **interchange_port_range** (*(int, int)*) – Port range used by Parsl to communicate with the Interchange.
- **working_dir** (*str*) – Working dir to be used by the executor.
- **worker_debug** (*Bool*) – Enables worker debug logging.
- **managed** (*Bool*) – If this executor is managed by the DFK or externally handled.
- **cores_per_worker** (*float*) – cores to be assigned to each worker. Oversubscription is possible by setting cores_per_worker < 1.0. Default=1
- **mem_per_worker** (*float*) – GB of memory required per worker. If this option is specified, the node manager will check the available memory at startup and limit the number of workers such that the there's sufficient memory for each worker. Default: None
- **max_workers** (*int*) – Caps the number of workers launched by the manager. Default: infinity
- **prefetch_capacity** (*int*) – Number of tasks that could be prefetched over available worker capacity. When there are a few tasks (<100) or when tasks are long running, this option should be set to 0 for better load balancing. Default is 0.

- **suppress_failure** (*Bool*) – If set, the interchange will suppress failures rather than terminate early. Default: True
- **heartbeat_threshold** (*int*) – Seconds since the last message from the counterpart in the communication pair: (interchange, manager) after which the counterpart is assumed to be un-available. Default: 120s
- **heartbeat_period** (*int*) – Number of seconds after which a heartbeat message indicating liveness is sent to the counterpart (interchange, manager). Default: 30s
- **poll_period** (*int*) – Timeout period to be used by the executor components in milliseconds. Increasing poll_periods trades performance for cpu efficiency. Default: 10ms
- **worker_logdir_root** (*string*) – In case of a remote file system, specify the path to where logs will be kept.

__init__(*label: str = 'HighThroughputExecutor', provider: parsl.providers.provider_base.ExecutionProvider = LocalProvider(channel=LocalChannel(envs={}, script_dir=None, userhome='/home/docs/checkouts/readthedocs.org/user_builds/parsl/checkouts/0.9.0/docs'), cmd_timeout=30, init_blocks=4, launcher=SingleNodeLauncher(), max_blocks=10, min_blocks=0, move_files=None, nodes_per_block=1, parallelism=1, walltime='00:15:00', worker_init=''), launch_cmd: Optional[str] = None, address: str = '127.0.0.1', worker_ports: Optional[Tuple[int, int]] = None, worker_port_range: Optional[Tuple[int, int]] = (54000, 55000), interchange_port_range: Optional[Tuple[int, int]] = (55000, 56000), storage_access: Optional[List[parsl.data_provider.staging.Staging]] = None, working_dir: Optional[str] = None, worker_debug: bool = False, cores_per_worker: float = 1.0, mem_per_worker: Optional[float] = None, max_workers: Union[int, float] = inf, prefetch_capacity: int = 0, heartbeat_threshold: int = 120, heartbeat_period: int = 30, poll_period: int = 10, suppress_failure: bool = True, managed: bool = True, worker_logdir_root: Optional[str] = None*)

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

_start_local_queue_process()

Starts the interchange process locally

Starts the interchange process locally and uses an internal command queue to get the worker task and result ports that the interchange has bound to.

_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.

hold_worker(*worker_id*)

Puts a worker on hold, preventing scheduling of additional tasks to it.

This is called "hold" mostly because this only stops scheduling of tasks, and does not actually kill the worker.

Parameters **worker_id** (*str*) – Worker id to be put on hold

scale_in(*blocks=None, block_ids=[]*)

Scale in the number of active blocks by specified amount.

The scale in method here is very rude. It doesn't give the workers the opportunity to finish current tasks or cleanup. This is tracked in issue #530

Parameters

- **blocks** (*int*) – Number of blocks to terminate and scale_in by
- **block_ids** (*list*) – List of specific block ids to terminate. Optional

- **Raises** – NotImplementedError

scale_out (*blocks=1*)
Scales out the number of blocks by "blocks"

Raises NotImplementedError

scaling_enabled
Specify if scaling is enabled.

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

start ()
Create the Interchange process and connect to it.

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 behaves like a submit call as described here Python docs:

Parameters

- **func** (–) – Callable function
- ***args** (–) – List of arbitrary positional arguments.

Kwargs:

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

Returns Future

WorkQueueExecutor

class parsl.executors.**WorkQueueExecutor** (*label='WorkQueueExecutor'*, *working_dir='.'*, *managed=True*, *project_name=None*, *project_password=None*, *project_password_file=None*, *port=0*, *env=None*, *shared_fs=False*, *source=False*, *init_command="*, *full_debug=True*, *see_worker_output=False*)
Executor to use Work Queue batch system

The WorkQueueExecutor system utilizes the Work Queue framework to efficiently delegate Parsl apps to remote machines in clusters and grids using a fault-tolerant system. Users can run the work_queue_worker program on remote machines to connect to the WorkQueueExecutor, and Parsl apps will then be sent out to these machines for execution and retrieval.

label: str A human readable label for the executor, unique with respect to other Work Queue master programs

working_dir: str Location for Parsl to perform app delegation to the Work Queue system

managed: bool If this executor is managed by the DFK or externally handled

project_name: str Work Queue process name

project_password: str Optional password for the Work Queue project

project_password_file: str Optional password file for the work queue project

port: int TCP port on Parsl submission machine for Work Queue workers to connect to. Workers will specify this port number when trying to connect to Parsl

env: dict{str} Dictionary that contains the environmental variables that need to be set on the Work Queue worker machine

shared_fs: bool Define if working in a shared file system or not. If Parsl and the Work Queue workers are on a shared file system, Work Queue does not need to transfer and rename files for execution

source: bool Choose whether to transfer parsl app information as source code. (Note: this increases throughput for @python_apps, but the implementation does not include functionality for @bash_apps, and thus source=False must be used for programs utilizing @bash_apps.)

init_command: str Command to run before constructed Work Queue command

see_worker_output: bool Prints worker standard output if true

`__init__`(*label='WorkQueueExecutor', working_dir='.', managed=True, project_name=None, project_password=None, project_password_file=None, port=0, env=None, shared_fs=False, source=False, init_command='', full_debug=True, see_worker_output=False*)
Initialize self. See help(type(self)) for accurate signature.

`create_name_tuple`(*parsl_file_obj, in_or_out*)
Returns a tuple containing information about an input or output file to a Parsl app. Utilized to specify input and output files for a specific Work Queue task within the system.

Parameters

- **`parsl_file_obj`** (`str`) – Name of file specified as input or output file to the Parsl app
- **`in_or_out`** (`str`) – Specifies whether the file is an input or output file to the Parsl app

`create_new_name`(*file_name*)
Returns a unique file name for an input file name. If the file name is already unique with respect to the Parsl process, then it returns the original file name

Parameters **`file_name`** (`str`) – Name of file that needs to be unique

`run_dir`(*value=None*)
Path to the run directory.

`scale_in`(*count*)
Scale in method. Not implemented.

`scale_out`(**args, **kwargs*)
Scale out method. Not implemented.

`scaling_enabled`()
Specify if scaling is enabled. Not enabled in Work Queue.

`shutdown`(**args, **kwargs*)
Shutdown the executor. Sets flag to cancel the submit process and collector thread, which shuts down the Work Queue system submission.

`start`()
Create submit process and collector thread to create, send, and retrieve Parsl tasks within the Work Queue system.

`submit`(*func, *args, **kwargs*)
Processes the Parsl app by its arguments and submits the function information to the task queue, to be executed using the Work Queue system. The args and kwargs are processed for input and output files to the Parsl app, so that the files are appropriately specified for the Work Queue task.

Parameters

- **func** (*function*) – Parsl app to be submitted to the Work Queue system
- **args** (*list*) – Arguments to the Parsl app
- **kwargs** (*dict*) – Keyword arguments to the Parsl app

ExtremeScaleExecutor

class parsl.executors.**ExtremeScaleExecutor**(*label='ExtremeScaleExecutor', provider=LocalProvider(channel=LocalChannel(envs={}, script_dir=None, userhome='/home/docs/checkouts/readthedocs.org/user_builds/parsl/check), cmd_timeout=30, init_blocks=4, launcher=SingleNodeLauncher(), max_blocks=10, min_blocks=0, move_files=None, nodes_per_block=1, parallelism=1, walltime='00:15:00', worker_init=''), launch_cmd=None, address='127.0.0.1', worker_ports=None, worker_port_range=(54000, 55000), interchange_port_range=(55000, 56000), storage_access=None, working_dir=None, worker_debug=False, ranks_per_node=1, heartbeat_threshold=120, heartbeat_period=30, managed=True*)

Executor designed for leadership class supercomputer scale

The ExtremeScaleExecutor extends the Executor interface to enable task execution on supercomputing systems (>1K Nodes). When functions and their arguments are submitted to the interface, a future is returned that tracks the execution of the function on a distributed compute environment.

The ExtremeScaleExecutor system has the following components:

1. The ExtremeScaleExecutor instance which is run as part of the Parsl script
2. The Interchange which is acts as a load-balancing proxy between workers and Parsl
3. The MPI based mpi_worker_pool which coordinates task execution over several nodes With MPI communication between workers, we can exploit low latency networking on HPC systems.
4. ZeroMQ pipes that connect the ExtremeScaleExecutor, Interchange and the mpi_worker_pool

Our design assumes that there is a single MPI application (mpi_worker_pool) running over a `block` and that there might be several such instances.

Here is a diagram

```
             |  Data   |  Executor    |  Interchange  | External Process(es)
             |  Flow   |              |               |
        Task | Kernel  |              |               |
      +----->|-------->|------------->|->outgoing_q---|-> mpi_worker_pool
      |      |         |              | batching      |    |         |
Parsl<---Fut-|         |              | load-balancing|  result   exception
          ^  |         |              | watchdogs     |    |         |
          |  |         |   Q_mngmnt   |               |    V         V
          |  |         |    Thread<--|-incoming_q<---|--- +---------+
```

(continues on next page)

(continued from previous page)

```
            |  |          |       |       |                |
            |  |          |       |       |                |
            +----update_fut-----+
```

Parameters

- **provider** (*ExecutionProvider*) –

  Provider to access computation resources. Can be any providers in parsl.providers:
  `Cobalt`, `Condor`, `GoogleCloud`, `GridEngine`, `Jetstream`, `Local`, `GridEngine`, `Slurm`, or `Torque`.
- **label** (*str*) – Label for this executor instance.
- **launch_cmd** (*str*) – Command line string to launch the mpi_worker_pool from the provider. The command line string will be formatted with appropriate values for the following values (debug, task_url, result_url, ranks_per_node, nodes_per_block, heartbeat_period ,heartbeat_threshold, logdir). For example: launch_cmd="mpiexec -np {ranks_per_node} mpi_worker_pool.py {debug} –task_url={task_url} –result_url={result_url}"
- **address** (*string*) – An address to connect to the main Parsl process which is reachable from the network in which workers will be running. This can be either a hostname as returned by `hostname` or an IP address. Most login nodes on clusters have several network interfaces available, only some of which can be reached from the compute nodes. Some trial and error might be necessary to identify what addresses are reachable from compute nodes.
- **worker_ports** (*(int, int)*) – Specify the ports to be used by workers to connect to Parsl. If this option is specified, worker_port_range will not be honored.
- **worker_port_range** (*(int, int)*) – Worker ports will be chosen between the two integers provided.
- **interchange_port_range** (*(int, int)*) – Port range used by Parsl to communicate with the Interchange.
- **working_dir** (*str*) – Working dir to be used by the executor.
- **worker_debug** (*Bool*) – Enables engine debug logging.
- **managed** (*Bool*) – If this executor is managed by the DFK or externally handled.
- **ranks_per_node** (*int*) – Specify the ranks to be launched per node.
- **heartbeat_threshold** (*int*) – Seconds since the last message from the counterpart in the communication pair: (interchange, manager) after which the counterpart is assumed to be un-available. Default:120s
- **heartbeat_period** (*int*) – Number of seconds after which a heartbeat message indicating liveness is sent to the counterpart (interchange, manager). Default:30s

__init__(*label='ExtremeScaleExecutor', provider=LocalProvider(channel=LocalChannel(envs={}, script_dir=None, userhome='/home/docs/checkouts/readthedocs.org/user_builds/parsl/checkouts/0.9.0/docs'), cmd_timeout=30, init_blocks=4, launcher=SingleNodeLauncher(), max_blocks=10, min_blocks=0, move_files=None, nodes_per_block=1, parallelism=1, walltime='00:15:00', worker_init=''), launch_cmd=None, address='127.0.0.1', worker_ports=None, worker_port_range=(54000, 55000), interchange_port_range=(55000, 56000), storage_access=None, working_dir=None, worker_debug=False, ranks_per_node=1, heartbeat_threshold=120, heartbeat_period=30, managed=True*)

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

`_start_local_queue_process`()
Starts the interchange process locally

Starts the interchange process locally and uses an internal command queue to get the worker task and result ports that the interchange has bound to.

`_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.

`hold_worker`(*worker_id*)
Puts a worker on hold, preventing scheduling of additional tasks to it.

This is called "hold" mostly because this only stops scheduling of tasks, and does not actually kill the worker.

Parameters **`worker_id`** (`str`) – Worker id to be put on hold

`scale_in`(*blocks=None*, *block_ids=[]*)
Scale in the number of active blocks by specified amount.

The scale in method here is very rude. It doesn't give the workers the opportunity to finish current tasks or cleanup. This is tracked in issue #530

Parameters

- **`blocks`** (`int`) – Number of blocks to terminate and scale_in by
- **`block_ids`** (`list`) – List of specific block ids to terminate. Optional
- **`Raises`** – NotImplementedError

`scale_out`(*blocks=1*)
Scales out the number of blocks by "blocks"

Raises `NotImplementedError`

`scaling_enabled`
Specify if scaling is enabled.

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

`start`()
Create the Interchange process and connect to it.

`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 behaves like a submit call as described here Python docs:

Parameters

- **`func`** (*–*) – Callable function
- **`*args`** (*–*) – List of arbitrary positional arguments.

Kwargs:

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

Returns Future

Swift/Turbine Executor

class parsl.executors.swift_t.**TurbineExecutor**(*label='turbine'*, *storage_access=None*, *working_dir=None*, *managed=True*)

The Turbine executor.

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

Here is a diagram

```
             |  Data   |  Executor   |   IPC      | External Process(es)
             |  Flow   |             |            |
        Task | Kernel  |             |            |
      +----->|-------->|------------>|outgoing_q -|-> Worker_Process
      |      |         |             |            |    |         |
Parsl<---Fut-|         |             |            |  result   exception
          ^  |         |             |            |    |         |
          |  |         |   Q_mngmnt  |            |    V         V
          |  |         |    Thread<--|incoming_q<-|--- +---------+
          |  |         |      |      |            |
          |  |         |      |      |            |
          +----update_fut-----+
```

__init__(*label='turbine'*, *storage_access=None*, *working_dir=None*, *managed=True*)

Initialize the thread pool.

Trying to implement the emews model.

_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(*blocks*)
Scale in the number of active blocks by specified amount.

This method is not implemented for turbine and will raise an error if called.

Raises NotImplementedError

scale_out(*blocks=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:

Parameters

- **func** (–) – Callable function
- ***args** (–) – 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.

Parameters

- **incoming_q** (–) – The queue to listen on
- **outgoing_q** (–) – 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

6.6.9 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 `parsl.providers.provider_base.`**`ExecutionProvider`**

Define the strict interface for all Execution Providers

```
                        +------------------
                        |
script_string ------->|  submit
     id      <--------|---+
                        |
[ ids ]       ------->|  status
[statuses]   <--------|----+
                        |
[ ids ]       ------->|  cancel
[cancel]     <--------|----+
                        |
                        +-------------------
```

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

`cancel`(*job_ids: List[Any]*) → List[bool]
Cancels the resources identified by the job_ids provided by the user.

Parameters **`job_ids`** (–) – A list of job identifiers

Returns

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

Raises

- ExecutionProviderException or its subclasses

`cores_per_node`
Number of cores to provision per node.

Providers which set this property should ask for cores_per_node cores when provisioning resources, and set the corresponding environment variable PARSL_CORES before executing submitted commands.

If this property is set, executors may use it to calculate how many tasks can run concurrently per node. This information is used by dataflow.Strategy to estimate the resources required to run all outstanding tasks.

label
Provides the label for this provider

mem_per_node
Real memory to provision per node in GB.

Providers which set this property should ask for mem_per_node of memory when provisioning resources, and set the corresponding environment variable PARSL_MEMORY_GB before executing submitted commands.

If this property is set, executors may use it to calculate how many tasks can run concurrently per node. This information is used by dataflow.Strategy to estimate the resources required to run all outstanding tasks.

status (*job_ids: List[Any]*) → List[str]
Get the status of a list of jobs identified by the job identifiers returned from the submit request.

Parameters **job_ids** (–) – 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

- ExecutionProviderException or its subclasses

submit (*command: str*, *tasks_per_node: int*, *job_name: str = 'parsl.auto'*) → Any
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 :

- command (str) : The bash command string to be executed
- tasks_per_node (int) : command invocations to be launched per node

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 or None or any other object that evaluates to boolean false

  if submission failed but an exception isn't thrown.

Raises

- ExecutionProviderException or its subclasses

Local

class parsl.providers.**LocalProvider** (*channel=LocalChannel(envs={}, script_dir=None, userhome='/home/docs/checkouts/readthedocs.org/user_builds/parsl/checkouts/0.9.0/), nodes_per_block=1, launcher=SingleNodeLauncher(), init_blocks=4, min_blocks=0, max_blocks=10, walltime='00:15:00', worker_init='', cmd_timeout=30, parallelism=1, move_files=None*)
Local Execution Provider

This provider is used to provide execution resources from the localhost.

Parameters

- **`min_blocks`** (*`int`*) – Minimum number of blocks to maintain.
- **`max_blocks`** (*`int`*) – Maximum number of blocks to maintain.
- **`parallelism`** (*`float`*) – Ratio of provisioned task slots to active tasks. 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., min_blocks) are used.
- **`move_files`** (*`Optional[Bool]: should files be moved? by default, Parsl will try to figure`*) – this out itself (= None). If True, then will always move. If False, will never move.
- **`worker_init`** (*`str`*) – Command to be run before starting a worker, such as 'module load Anaconda; source activate env'.

`__init__`(*channel=LocalChannel(envs={}, script_dir=None, userhome='/home/docs/checkouts/readthedocs.org/user_builds/parsl/checkouts/0.9.0/docs'), nodes_per_block=1, launcher=SingleNodeLauncher(), init_blocks=4, min_blocks=0, max_blocks=10, walltime='00:15:00', worker_init='', cmd_timeout=30, parallelism=1, move_files=None*)

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

`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.

`label`

Provides the label for this provider

`status`(*job_ids*)

Get the status of a list of jobs identified by their ids.

Parameters **`job_ids`** (*–*) – List of identifiers for the jobs

Returns

- List of status codes.

`submit`(*command, tasks_per_node, job_name='parsl.auto'*)

Submits the command onto an Local Resource Manager job. 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 nodes are provisioned.

Parameters

- **`command`** (*–*) – (String) Commandline invocation to be made on the remote side.
- **`tasks_per_node`** (*–*) – command invocations to be launched per node

Kwargs:

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

Returns At capacity, cannot provision more - job_id: (string) Identifier for the job

Return type

- None

Slurm

class parsl.providers.**SlurmProvider**(*partition*, *channel=LocalChannel(envs={}*, *script_dir=None*, *user-home='/home/docs/checkouts/readthedocs.org/user_builds/parsl/checkouts/0.9.0/)*, *nodes_per_block=1*, *cores_per_node=None*, *mem_per_node=None*, *init_blocks=1*, *min_blocks=0*, *max_blocks=10*, *parallelism=1*, *walltime='00:10:00'*, *scheduler_options="*, *worker_init="*, *cmd_timeout=10*, *exclusive=True*, *move_files=True*, *launcher=SingleNodeLauncher()*)

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.

Parameters

- **partition** (*str*) – Slurm partition to request blocks from.
- **channel** (Channel) – Channel for accessing this provider. Possible channels include *LocalChannel* (the default), *SSHChannel*, or *SSHInteractiveLoginChannel*.
- **nodes_per_block** (*int*) – Nodes to provision per block.
- **cores_per_node** (*int*) – Specify the number of cores to provision per node. If set to None, executors will assume all cores on the node are available for computation. Default is None.
- **mem_per_node** (*float*) – Specify the real memory to provision per node in GB. If set to None, no explicit request to the scheduler will be made. Default is None.
- **min_blocks** (*int*) – Minimum number of blocks to maintain.
- **max_blocks** (*int*) – Maximum number of blocks to maintain.
- **parallelism** (*float*) – Ratio of provisioned task slots to active tasks. 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., min_blocks) are used.
- **walltime** (*str*) – Walltime requested per block in HH:MM:SS.
- **scheduler_options** (*str*) – String to prepend to the #SBATCH blocks in the submit script to the scheduler.
- **worker_init** (*str*) – Command to be run before starting a worker, such as 'module load Anaconda; source activate env'.
- **exclusive** (*bool (Default = True)*) – Requests nodes which are not shared with other running jobs.
- **launcher** (*Launcher*) –

  Launcher for this provider. Possible launchers include *SingleNodeLauncher* (the default), *SrunLauncher*, or *AprunLauncher* move_files : Optional[Bool]: should files be moved? by default, Parsl will try to move files.

`__init__`(*partition, channel=LocalChannel(envs={}, script_dir=None, userhome='/home/docs/checkouts/readthedocs.org/user_builds/parsl/checkouts/0.9.0/docs'), nodes_per_block=1, cores_per_node=None, mem_per_node=None, init_blocks=1, min_blocks=0, max_blocks=10, parallelism=1, walltime='00:10:00', scheduler_options='', worker_init='', cmd_timeout=10, exclusive=True, move_files=True, launcher=SingleNodeLauncher()*)

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

`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.

`submit`(*command, tasks_per_node, job_name='parsl.auto'*)

Submit the command as a slurm job.

Parameters

- **`command`** (`str`) – Command to be made on the remote side.
- **`tasks_per_node`** (`int`) – Command invocations to be launched per node
- **`job_name`** (`str`) – Name for the job (must be unique).

Returns If at capacity, returns None; otherwise, a string identifier for the job

Return type None or str

Cobalt

class `parsl.providers.`**`CobaltProvider`**(*channel=LocalChannel(envs={}, script_dir=None, userhome='/home/docs/checkouts/readthedocs.org/user_builds/parsl/checkouts/0.9.0), nodes_per_block=1, init_blocks=0, min_blocks=0, max_blocks=10, parallelism=1, walltime='00:10:00', account=None, queue=None, scheduler_options='', worker_init='', launcher=AprunLauncher(overrides=''), cmd_timeout=10*)

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.

Parameters

- **`channel`** (`Channel`) – Channel for accessing this provider. Possible channels include *`LocalChannel`* (the default), *`SSHChannel`*, or *`SSHInteractiveLoginChannel`*.
- **`nodes_per_block`** (`int`) – Nodes to provision per block.
- **`min_blocks`** (`int`) – Minimum number of blocks to maintain.
- **`max_blocks`** (`int`) – Maximum number of blocks to maintain.
- **`walltime`** (`str`) – Walltime requested per block in HH:MM:SS.
- **`account`** (`str`) – Account that the job will be charged against.
- **`queue`** (`str`) – Torque queue to request blocks from.

- **scheduler_options** (`str`) – String to prepend to the submit script to the scheduler.
- **worker_init** (`str`) – Command to be run before starting a worker, such as 'module load Anaconda; source activate env'.
- **launcher** (`Launcher`) – Launcher for this provider. Possible launchers include `AprunLauncher` (the default) or, `SingleNodeLauncher`

`__init__`(*channel=LocalChannel(envs={}, script_dir=None, user-home='/home/docs/checkouts/readthedocs.org/user_builds/parsl/checkouts/0.9.0/docs'), nodes_per_block=1, init_blocks=0, min_blocks=0, max_blocks=10, parallelism=1, walltime='00:10:00', account=None, queue=None, scheduler_options=", worker_init=", launcher=AprunLauncher(overrides="), cmd_timeout=10*)

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

`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.

`submit`(*command, tasks_per_node, job_name='parsl.auto'*)

Submits the command onto an Local Resource Manager job of 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 number of nodes are provisioned.

Parameters

- **command** (–) – (String) Commandline invocation to be made on the remote side.
- **tasks_per_node** (–) – command invocations to be launched per node

Kwargs:

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

Returns At capacity, cannot provision more - job_id: (string) Identifier for the job

Return type

- None

Condor

class parsl.providers.**CondorProvider**(*channel: parsl.channels.base.Channel = LocalChannel(envs={}, script_dir=None, userhome='/home/docs/checkouts/readthedocs.org/user_builds/parsl/checkouts/0.9.0), nodes_per_block: int = 1, cores_per_slot: Optional[int] = None, mem_per_slot: Optional[float] = None, init_blocks: int = 1, min_blocks: int = 0, max_blocks: int = 10, parallelism: float = 1, environment: Optional[Dict[str, str]] = None, project: str = ", scheduler_options: str = ", transfer_input_files: List[str] = [], walltime: str = '00:10:00', worker_init: str = ", launcher: parsl.launchers.launchers.Launcher = SingleNodeLauncher(), requirements: str = ", cmd_timeout: int = 60*)

HTCondor Execution Provider.

Parameters

- **channel** (Channel) – Channel for accessing this provider. Possible channels include *LocalChannel* (the default), *SSHChannel*, or *SSHInteractiveLoginChannel*.
- **nodes_per_block** (*int*) – Nodes to provision per block.
- **cores_per_slot** (*int*) – Specify the number of cores to provision per slot. If set to None, executors will assume all cores on the node are available for computation. Default is None.
- **mem_per_slot** (*float*) – Specify the real memory to provision per slot in GB. If set to None, no explicit request to the scheduler will be made. Default is None.
- **init_blocks** (*int*) – Number of blocks to provision at time of initialization
- **min_blocks** (*int*) – Minimum number of blocks to maintain
- **max_blocks** (*int*) – Maximum number of blocks to maintain.
- **parallelism** (*float*) – Ratio of provisioned task slots to active tasks. 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., min_blocks) are used.
- **environment** (*dict of str*) – A dictionary of environmant variable name and value pairs which will be set before running a task.
- **project** (*str*) – Project which the job will be charged against
- **scheduler_options** (*str*) – String to add specific condor attributes to the HTCondor submit script.
- **transfer_input_files** (*list(str)*) – List of strings of paths to additional files or directories to transfer to the job
- **worker_init** (*str*) – Command to be run before starting a worker.
- **requirements** (*str*) – Condor requirements.
- **launcher** (*Launcher*) – Launcher for this provider. Possible launchers include *SingleNodeLauncher* (the default),
- **cmd_timeout** (*int*) – Timeout for commands made to the scheduler in seconds

`__init__`(*channel: parsl.channels.base.Channel = LocalChannel(envs={}, script_dir=None, user-home='/home/docs/checkouts/readthedocs.org/user_builds/parsl/checkouts/0.9.0/docs'), nodes_per_block: int = 1, cores_per_slot: Optional[int] = None, mem_per_slot: Optional[float] = None, init_blocks: int = 1, min_blocks: int = 0, max_blocks: int = 10, parallelism: float = 1, environment: Optional[Dict[str, str]] = None, project: str = '', scheduler_options: str = '', transfer_input_files: List[str] = [], walltime: str = '00:10:00', worker_init: str = '', launcher: parsl.launchers.launchers.Launcher = SingleNodeLauncher(), requirements: str = '', cmd_timeout: int = 60*) → None

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

`cancel`(*job_ids*)

Cancels the jobs specified by a list of job IDs.

Parameters **`job_ids`** (*`list of str`*) – The job IDs to cancel.

Returns Each entry in the list will be True if the job is cancelled succesfully, otherwise False.

Return type list of bool

`current_capacity`

Returns the currently provisioned blocks. This may need to return more information in the futures : { minsize, maxsize, current_requested }

`status`(*job_ids*)

Get the status of a list of jobs identified by their ids.

Parameters **`job_ids`** (*`list of int`*) – Identifiers of jobs for which the status will be returned.

Returns Status codes for the requested jobs.

Return type List of int

`submit`(*command, tasks_per_node, job_name='parsl.auto'*)

Submits the command onto an Local Resource Manager job.

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.

Parameters

- **`command`** (*`str`*) – Command to execute
- **`job_name`** (*`str`*) – Job name prefix.
- **`tasks_per_node`** (*`int`*) – command invocations to be launched per node

Returns None if at capacity and cannot provision more; otherwise the identifier for the job.

Return type None or str

Torque

class parsl.providers.**TorqueProvider**(*channel=LocalChannel(envs={}, script_dir=None, user-home='/home/docs/checkouts/readthedocs.org/user_builds/parsl/checkouts/0.9.0), account=None, queue=None, scheduler_options='', worker_init='', nodes_per_block=1, init_blocks=1, min_blocks=0, max_blocks=100, parallelism=1, launcher=AprunLauncher(overrides=''), walltime='00:20:00', cmd_timeout=120*)

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.

Parameters

- **channel** (Channel) – Channel for accessing this provider. Possible channels include *LocalChannel* (the default), *SSHChannel*, or *SSHInteractiveLoginChannel*.
- **account** (*str*) – Account the job will be charged against.
- **queue** (*str*) – Torque queue to request blocks from.
- **nodes_per_block** (*int*) – Nodes to provision per block.
- **init_blocks** (*int*) – Number of blocks to provision at the start of the run. Default is 1.
- **min_blocks** (*int*) – Minimum number of blocks to maintain. Default is 0.
- **max_blocks** (*int*) – Maximum number of blocks to maintain.
- **parallelism** (*float*) – Ratio of provisioned task slots to active tasks. 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., min_blocks) are used.
- **walltime** (*str*) – Walltime requested per block in HH:MM:SS.
- **scheduler_options** (*str*) – String to prepend to the #PBS blocks in the submit script to the scheduler.
- **worker_init** (*str*) – Command to be run before starting a worker, such as 'module load Anaconda; source activate env'.
- **launcher** (*Launcher*) – Launcher for this provider. Possible launchers include *AprunLauncher* (the default), or *SingleNodeLauncher*,

__init__(*channel=LocalChannel(envs={}, script_dir=None, user-home='/home/docs/checkouts/readthedocs.org/user_builds/parsl/checkouts/0.9.0/docs'), account=None, queue=None, scheduler_options='', worker_init='', nodes_per_block=1, init_blocks=1, min_blocks=0, max_blocks=100, parallelism=1, launcher=AprunLauncher(overrides=''), walltime='00:20:00', cmd_timeout=120*)

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

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.

submit(*command*, *tasks_per_node*, *job_name='parsl.auto'*)

Submits the command onto an Local Resource Manager job. 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 number of nodes are provisioned.

Parameters

- **command** (–) – (String) Commandline invocation to be made on the remote side.
- **tasks_per_node** (–) – command invocations to be launched per node

Kwargs:

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

Returns At capacity, cannot provision more - job_id: (string) Identifier for the job

Return type

- None

GridEngine

class parsl.providers.**GridEngineProvider**(*channel=LocalChannel(envs={}, script_dir=None, userhome='/home/docs/checkouts/readthedocs.org/user_builds/parsl/checkout), nodes_per_block=1, init_blocks=1, min_blocks=0, max_blocks=10, parallelism=1, walltime='00:10:00', scheduler_options='', worker_init='', launcher=SingleNodeLauncher()*)

A provider for the Grid Engine scheduler.

Parameters

- **channel** (Channel) – Channel for accessing this provider. Possible channels include *LocalChannel* (the default), *SSHChannel*, or *SSHInteractiveLoginChannel*.
- **nodes_per_block** (*int*) – Nodes to provision per block.
- **min_blocks** (*int*) – Minimum number of blocks to maintain.
- **max_blocks** (*int*) – Maximum number of blocks to maintain.
- **parallelism** (*float*) – Ratio of provisioned task slots to active tasks. 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., min_blocks) are used.
- **walltime** (*str*) – Walltime requested per block in HH:MM:SS.
- **scheduler_options** (*str*) – String to prepend to the #$$ blocks in the submit script to the scheduler.
- **worker_init** (*str*) – Command to be run before starting a worker, such as 'module load Anaconda; source activate env'.

- **launcher** (*Launcher*) – Launcher for this provider. Possible launchers include *SingleNodeLauncher* (the default),

__init__(*channel=LocalChannel(envs={}, script_dir=None, userhome='/home/docs/checkouts/readthedocs.org/user_builds/parsl/checkouts/0.9.0/docs'), nodes_per_block=1, init_blocks=1, min_blocks=0, max_blocks=10, parallelism=1, walltime='00:10:00', scheduler_options='', worker_init='', launcher=SingleNodeLauncher()*)
Initialize self. See help(type(self)) for accurate signature.

cancel(*job_ids*)
Cancels the resources identified by the job_ids provided by the user.

Parameters **job_ids** (*-*) – A list of job identifiers

Returns

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

Raises

- ExecutionProviderException or its subclasses

get_configs(*command, tasks_per_node*)
Compose a dictionary with information for writing the submit script.

submit(*command, tasks_per_node, 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 :

- command (str) : The bash command string to be executed.
- tasks_per_node (int) : command invocations to be launched per node

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

- ExecutionProviderException or its subclasses

Amazon Web Services

class parsl.providers.**AWSProvider**(*image_id, key_name, init_blocks=1, min_blocks=0, max_blocks=10, nodes_per_block=1, parallelism=1, worker_init='', instance_type='t2.small', region='us-east-2', spot_max_bid=0, key_file=None, profile=None, iam_instance_profile_arn='', state_file=None, walltime='01:00:00', linger=False, launcher=SingleNodeLauncher()*)
A provider for using Amazon Elastic Compute Cloud (EC2) resources.

One of 3 methods are required to authenticate: keyfile, profile or environment variables. If neither keyfile or profile are set, the following environment variables must be set: `AWS_ACCESS_KEY_ID` (the access key for your AWS account), `AWS_SECRET_ACCESS_KEY` (the secret key for your AWS account), and (optionaly) the `AWS_SESSION_TOKEN` (the session key for your AWS account).

Parameters

- **image_id** (*str*) – Identification of the Amazon Machine Image (AMI).
- **worker_init** (*str*) – String to append to the Userdata script executed in the cloudinit phase of instance initialization.
- **walltime** (*str*) – Walltime requested per block in HH:MM:SS.
- **key_file** (*str*) – Path to json file that contains 'AWSAccessKeyId' and 'AWSSecretKey'.
- **nodes_per_block** (*int*) – This is always 1 for ec2. Nodes to provision per block.
- **profile** (*str*) – Profile to be used from the standard aws config file ~/.aws/config.
- **nodes_per_block** – Nodes to provision per block. Default is 1.
- **init_blocks** (*int*) – Number of blocks to provision at the start of the run. Default is 1.
- **min_blocks** (*int*) – Minimum number of blocks to maintain. Default is 0.
- **max_blocks** (*int*) – Maximum number of blocks to maintain. Default is 10.
- **instance_type** (*str*) – EC2 instance type. Instance types comprise varying combinations of CPU, memory, . storage, and networking capacity For more information on possible instance types,. see here Default is 't2.small'.
- **region** (*str*) – Amazon Web Service (AWS) region to launch machines. Default is 'us-east-2'.
- **key_name** (*str*) – Name of the AWS private key (.pem file) that is usually generated on the console to allow SSH access to the EC2 instances. This is mostly used for debugging.
- **spot_max_bid** (*float*) – Maximum bid price (if requesting spot market machines).
- **iam_instance_profile_arn** (*str*) – Launch instance with a specific role.
- **state_file** (*str*) – Path to the state file from a previous run to re-use.
- **walltime** – Walltime requested per block in HH:MM:SS. This option is not currently honored by this provider.
- **launcher** (*Launcher*) – Launcher for this provider. Possible launchers include *SingleNodeLauncher* (the default), *SrunLauncher*, or *AprunLauncher*
- **linger** (*Bool*) – When set to True, the workers will not `halt`. The user is responsible for shutting down the node.

__init__(*image_id*, *key_name*, *init_blocks=1*, *min_blocks=0*, *max_blocks=10*, *nodes_per_block=1*, *parallelism=1*, *worker_init=''*, *instance_type='t2.small'*, *region='us-east-2'*, *spot_max_bid=0*, *key_file=None*, *profile=None*, *iam_instance_profile_arn=''*, *state_file=None*, *walltime='01:00:00'*, *linger=False*, *launcher=SingleNodeLauncher()*)
Initialize self. See help(type(self)) for accurate signature.

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

Parameters **job_ids** (*list of str*) – List of of job identifiers

Returns Each entry in the list will contain False if the operation fails. Otherwise, the entry will be True.

Return type list of bool

config_route_table(*vpc*, *internet_gateway*)
Configure route table for Virtual Private Cloud (VPC).

Parameters

- **vpc** (*dict*) – Representation of the VPC (created by create_vpc()).
- **internet_gateway** (*dict*) – Representation of the internet gateway (created by create_vpc()).

create_session()
Create a session.

First we look in self.key_file for a path to a json file with the credentials. The key file should have 'AWSAccessKeyId' and 'AWSSecretKey'.

Next we look at self.profile for a profile name and try to use the Session call to automatically pick up the keys for the profile from the user default keys file ~/.aws/config.

Finally, boto3 will look for the keys in environment 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

We create a VPC with CIDR 10.0.0.0/16, which provides up to 64,000 instances.

We attach a subnet for each availability zone within the region specified in the config. We give each subnet an ip range like 10.0.X.0/20, which is large enough for approx. 4000 instances.

Security groups are configured in function security_group.

current_capacity
Returns the current blocksize.

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

initialize_boto_client()
Initialize the boto client.

label
Provides the label for this provider

read_state_file(*state_file*)
Read the state file, if it exists.

If this script has been run previously, resource IDs will have been written to a state file. On starting a run, a state file will be looked for before creating new infrastructure. Information on VPCs, security groups, and subnets are saved, as well as running instances and their states.

AWS has a maximum number of VPCs per region per account, so we do not want to clutter users' AWS accounts with security groups and VPCs that will be used only once.

security_group(*vpc*)
Create and configure a new security group.

Allows all ICMP in, all TCP and UDP in within VPC.

This security group is very open. It allows all incoming ping requests on all ports. It also allows all outgoing traffic on all ports. This can be limited by changing the allowed port ranges.

Parameters **vpc** (*VPC instance*) – VPC in which to set up security group.

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

shut_down_instance(*instances=None*)
Shut down a list of instances, if provided.

If no instance is provided, the last instance started up will be shut down.

spin_up_instance(*command*, *job_name*)
Start an instance in the VPC in the first available subnet.

N instances will be started if nodes_per_block > 1. Not supported. We only do 1 node per block.

Parameters

- **command** (`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.

Parameters **job_ids** (`list of str`) – Identifiers for the jobs.

Returns The status codes of the requsted jobs.

Return type list of int

submit(*command='sleep 1'*, *tasks_per_node=1*, *job_name='parsl.auto'*)
Submit the command onto a freshly instantiated AWS EC2 instance.

Submit returns an ID that corresponds to the task that was just submitted.

Parameters

- **command** (`str`) – Command to be invoked on the remote side.
- **tasks_per_node** (`int (default=1)`) – Number of command invocations to be launched per node
- **job_name** (`str`) – Prefix for the job name.

Returns If at capacity, None will be returned. Otherwise, the job identifier will be returned.

Return type None or str

teardown()
Teardown the EC2 infastructure.

Terminate all EC2 instances, delete all subnets, delete security group, delete VPC, and reset all instance variables.

write_state_file()
Save information that must persist to a file.

We do not want to create a new VPC and new identical security groups, so we save information about them in a file between runs.

Google Cloud Platform

class parsl.providers.**GoogleCloudProvider**(*project_id*, *key_file*, *region*, *os_project*, *os_family*, *google_version='v1'*, *instance_type='n1-standard-1'*, *init_blocks=1*, *min_blocks=0*, *max_blocks=10*, *launcher=SingleNodeLauncher()*, *parallelism=1*)

A provider for using resources from the Google Compute Engine.

Parameters

- **project_id** (*str*) – Project ID from Google compute engine.
- **key_file** (*str*) – Path to authorization private key json file. This is required for auth. A new one can be generated here: https://console.cloud.google.com/apis/credentials
- **region** (*str*) – Region in which to start instances
- **os_project** (*str*) – OS project code for Google compute engine.
- **os_family** (*str*) – OS family to request.
- **google_version** (*str*) – Google compute engine version to use. Possibilies include 'v1' (default) or 'beta'.
- **instance_type** (*str*) – 'n1-standard-1',
- **init_blocks** (*int*) – Number of blocks to provision immediately. Default is 1.
- **min_blocks** (*int*) – Minimum number of blocks to maintain. Default is 0.
- **max_blocks** (*int*) – Maximum number of blocks to maintain. Default is 10.
- **parallelism** (*float*) – Ratio of provisioned task slots to active tasks. 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., min_blocks) are used.

:param .. code:: python: +——————

script_string ——->| submit

id <———|—+

[ids] ——->| status [statuses] <———|—-+

[ids] ——->| cancel [cancel] <———|—-+

+———————-

__init__(*project_id*, *key_file*, *region*, *os_project*, *os_family*, *google_version='v1'*, *instance_type='n1-standard-1'*, *init_blocks=1*, *min_blocks=0*, *max_blocks=10*, *launcher=SingleNodeLauncher()*, *parallelism=1*)

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

cancel (*job_ids*)
Cancels the resources identified by the job_ids provided by the user.

Parameters **job_ids** (*–*) – A list of job identifiers

Returns

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

Raises

- ExecutionProviderException or its subclasses

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

Parameters **job_ids** (*–*) – 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

- ExecutionProviderException or its subclasses

submit (*command*, *tasks_per_node*, *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.

Args :

- command (str) : The bash command string to be executed.
- tasks_per_node (int) : command invocations to be launched per node

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

- ExecutionProviderException or its subclasses

Kubernetes

class parsl.providers.**KubernetesProvider** (*image: str*, *namespace: str = 'default'*, *nodes_per_block: int = 1*, *init_blocks: int = 4*, *min_blocks: int = 0*, *max_blocks: int = 10*, *max_cpu: float = 2*, *max_mem: str = '500Mi'*, *init_cpu: float = 1*, *init_mem: str = '250Mi'*, *parallelism: float = 1*, *worker_init: str = ''*, *pod_name: Optional[str] = None*, *user_id: Optional[str] = None*, *group_id: Optional[str] = None*, *run_as_non_root: bool = False*, *secret: Optional[str] = None*, *persistent_volumes: List[Tuple[str, str]] = []*)
Kubernetes execution provider :param namespace: Kubernetes namespace to create deployments. :type names-

pace: str :param image: Docker image to use in the deployment. :type image: str :param nodes_per_block: Nodes to provision per block. :type nodes_per_block: int :param init_blocks: Number of blocks to provision at the start of the run. Default is 1. :type init_blocks: int :param min_blocks: Minimum number of blocks to maintain. :type min_blocks: int :param max_blocks: Maximum number of blocks to maintain. :type max_blocks: int :param max_cpu: CPU limits of the blocks (pods), in cpu units.

This is the cpu "limits" option for resource specification. Check kubernetes docs for more details. Default is 2.

Parameters

- **max_mem** (*str*) – Memory limits of the blocks (pods), in Mi or Gi. This is the memory "limits" option for resource specification on kubernetes. Check kubernetes docs for more details. Default is 500Mi.
- **init_cpu** (*float*) – CPU limits of the blocks (pods), in cpu units. This is the cpu "requests" option for resource specification. Check kubernetes docs for more details. Default is 1.
- **init_mem** (*str*) – Memory limits of the blocks (pods), in Mi or Gi. This is the memory "requests" option for resource specification on kubernetes. Check kubernetes docs for more details. Default is 250Mi.
- **parallelism** (*float*) – Ratio of provisioned task slots to active tasks. 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., min_blocks) are used.
- **worker_init** (*str*) – Command to be run first for the workers, such as `python start.py`.
- **secret** (*str*) – Docker secret to use to pull images
- **pod_name** (*str*) – The name for the pod, will be appended with a timestamp. Default is None, meaning parsl automatically names the pod.
- **user_id** (*str*) – Unix user id to run the container as.
- **group_id** (*str*) – Unix group id to run the container as.
- **run_as_non_root** (*bool*) – Run as non-root (True) or run as root (False).
- **persistent_volumes** (*list[(str, str)]*) – List of tuples describing persistent volumes to be mounted in the pod. The tuples consist of (PVC Name, Mount Directory).

__init__(*image: str*, *namespace: str = 'default'*, *nodes_per_block: int = 1*, *init_blocks: int = 4*, *min_blocks: int = 0*, *max_blocks: int = 10*, *max_cpu: float = 2*, *max_mem: str = '500Mi'*, *init_cpu: float = 1*, *init_mem: str = '250Mi'*, *parallelism: float = 1*, *worker_init: str = ''*, *pod_name: Optional[str] = None*, *user_id: Optional[str] = None*, *group_id: Optional[str] = None*, *run_as_non_root: bool = False*, *secret: Optional[str] = None*, *persistent_volumes: List[Tuple[str, str]] = []*) → None

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

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.

label

Provides the label for this provider

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

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*, *tasks_per_node*, *job_name='parsl'*)
Submit a job :param - cmd_string: (String) - Name of the container to initiate :param - tasks_per_node: command invocations to be launched per node :type - tasks_per_node: int

Kwargs:

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

Returns At capacity, cannot provision more - job_id: (string) Identifier for the job

Return type

- None

6.6.10 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 parsl.channels.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.

```
                      +------------------
                      |
cmd, wtime    ------->|  execute_wait
(ec, stdout, stderr)<-|---+
                      |
cmd, wtime    ------->|  execute_no_wait
(ec, stdout, stderr)<-|---+
                      |
src, dst_dir  ------->|  push_file
   dst_path  <--------|----+
                      |
dst_script_dir <------|  script_dir
                      |
                      +-------------------
```

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

abspath (*path*)
Return the absolute path.

Parameters **path** (*str*) – Path for which the absolute path will be returned.

close()

Closes the channel. Clean out any auth credentials.

Parameters **None** –

Returns Bool

execute_no_wait(*cmd*, *walltime*, *envs={}*, **args*, ***kwargs*)

Execute asynchronousely without waiting for exitcode

Parameters

- **cmd** (*–*) – Command string to execute over the channel
- **walltime** (*–*) – Timeout in seconds

KWargs:

- envs (dict) : Environment variables to push to the remote side

Returns

- the type of return value is channel specific

execute_wait(*cmd*, *walltime=None*, *envs={}*, **args*, ***kwargs*)

Executes the cmd, with a defined walltime.

Parameters

- **cmd** (*–*) – Command string to execute over the channel
- **walltime** (*–*) – Timeout in seconds, optional

KWargs:

- envs (Dict[str, str]) : Environment variables to push to the remote side

Returns

- (exit_code, stdout, stderr) (int, optional string, optional string) If the exit code is a failure code, the stdout and stderr return values may be None.

isdir(*path*)

Return true if the path refers to an existing directory.

Parameters **path** (*str*) – Path of directory to check.

makedirs(*path*, *mode=511*, *exist_ok=False*)

Create a directory.

If intermediate directories do not exist, they will be created.

Parameters

- **path** (*str*) – Path of directory to create.
- **mode** (*int*) – Permissions (posix-style) for the newly-created directory.
- **exist_ok** (*bool*) – If False, raise an OSError if the target directory already exists.

push_file(*source*, *dest_dir*)

Channel will take care of moving the file from source to the destination directory

Parameters

- **`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.

Parameters **`None`** (*–*) –

Returns

- Channel script dir

LocalChannel

class `parsl.channels.`**`LocalChannel`**(*userhome='.'*, *envs={}*, *script_dir=None*, ***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={}*, *script_dir=None*, ***kwargs*)

Initialize the local channel. script_dir 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
- script_dir (string): Directory to place scripts

`abspath`(*path*)

Return the absolute path.

Parameters **`path`** (*`str`*) – Path for which the absolute path will be returned.

`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*, *envs={}*)

Synchronously execute a commandline string on the shell.

Parameters

- **`cmd`** (*–*) – Commandline string to execute
- **`walltime`** (*–*) – walltime in seconds, this is not really used now.

Returns a tuple containing:

- pid : process id
- proc : a subprocess.Popen object

Raises `None.`

execute_wait (*cmd*, *walltime=None*, *envs={}*)
Synchronously execute a commandline string on the shell.

Parameters

- **cmd** (–) – Commandline string to execute
- **walltime** (–) – walltime in seconds, this is not really used now.

Kwargs:

- envs (dict) : Dictionary of env variables. This will be used to override the envs set at channel initialization.

Returns Return code from the execution, -1 on fail - stdout : stdout string - stderr : stderr string

Return type

- retcode

Raises: None.

isdir (*path*)
Return true if the path refers to an existing directory.

Parameters path (*str*) – Path of directory to check.

makedirs (*path*, *mode=511*, *exist_ok=False*)
Create a directory.

If intermediate directories do not exist, they will be created.

Parameters

- **path** (*str*) – Path of directory to create.
- **mode** (*int*) – Permissions (posix-style) for the newly-created directory.
- **exist_ok** (*bool*) – If False, raise an OSError if the target directory already exists.

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.

Parameters

- **source** (–) – Path to the source file
- **dest_dir** (–) – Path to the directory to which the files is to be copied

Returns Absolute path of the destination file

Return type

- destination_path (String)

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.

Parameters None (–) –

Returns

- Channel script dir

SshChannel

class parsl.channels.**SSHChannel**(*hostname*, *username=None*, *password=None*, *script_dir=None*, *envs=None*, *gssapi_auth=False*, *skip_auth=False*, *port=22*, ***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*, *script_dir=None*, *envs=None*, *gssapi_auth=False*, *skip_auth=False*, *port=22*, ***kwargs*)

Initialize a persistent connection to the remote system. We should know at this point whether ssh connectivity is possible

Parameters **hostname** (*-*) – Hostname

KWargs:

- username (string) : Username on remote system
- password (string) : Password for remote system
- port : The port designated for the ssh connection. Default is 22.
- script_dir (string) : Full path to a script dir where generated scripts could be sent to.
- envs (dict) : A dictionary of environment variables to be set when executing commands

Raises:

abspath(*path*)

Return the absolute path on the remote side.

Parameters **path** (*str*) – Path for which the absolute path will be returned.

close()

Closes the channel. Clean out any auth credentials.

Parameters **None** –

Returns Bool

execute_no_wait(*cmd*, *walltime=2*, *envs={}*)

Execute asynchronousely without waiting for exitcode

Parameters

- **cmd** (*-*) – Commandline string to be executed on the remote side
- **walltime** (*-*) – timeout to exec_command

KWargs:

- envs (dict): A dictionary of env variables

Returns

- None, stdout (readable stream), stderr (readable stream)

Raises

- ChannelExecFailed (reason)

execute_wait(*cmd*, *walltime=2*, *envs={}*)
Synchronously execute a commandline string on the shell.

Parameters

- **cmd** (–) – Commandline string to execute
- **walltime** (–) – walltime in seconds

Kwargs:

- envs (dict) : Dictionary of env variables

Returns Return code from the execution, -1 on fail - stdout : stdout string - stderr : stderr string

Return type

- retcode

Raises: None.

isdir(*path*)
Return true if the path refers to an existing directory.

Parameters **path** (*str*) – Path of directory on the remote side to check.

makedirs(*path*, *mode=511*, *exist_ok=False*)
Create a directory on the remote side.

If intermediate directories do not exist, they will be created.

Parameters

- **path** (*str*) – Path of directory on the remote side to create.
- **mode** (*int*) – Permissions (posix-style) for the newly-created directory.
- **exist_ok** (*bool*) – If False, raise an OSError if the target directory already exists.

pull_file(*remote_source*, *local_dir*)
Transport file on the remote side to a local directory

Parameters

- **remote_source** (–) – remote_source
- **local_dir** (–) – Local directory to copy to

Returns Local path to file

Return type

- str

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

Parameters

- **local_source** (–) – Path
- **remote_dir** (–) – Remote path

Returns Path to copied file on remote machine

Return type

- str

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.

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.

Parameters **None** (*-*) –

Returns

- Channel script dir

SSH Interactive Login Channel

class parsl.channels.**SSHInteractiveLoginChannel**(*hostname*, *username=None*, *password=None*, *script_dir=None*, *envs=None*, ***kwargs*)

SSH persistent channel. This enables remote execution on sites accessible via ssh. This channel supports interactive login and is appropriate when keys are not set up.

__init__(*hostname*, *username=None*, *password=None*, *script_dir=None*, *envs=None*, ***kwargs*)

Initialize a persistent connection to the remote system. We should know at this point whether ssh connectivity is possible

Parameters **hostname** (*-*) – Hostname

KWargs:

- username (string) : Username on remote system
- password (string) : Password for remote system
- script_dir (string) : Full path to a script dir where generated scripts could be sent to.
- envs (dict) : A dictionary of env variables to be set when executing commands

Raises:

ExecutionProviders

An execution provider is basically an adapter to various types of execution resources. The providers abstract away the interfaces provided by various systems to request, monitor, and cancel computate resources.

Slurm

class parsl.providers.**SlurmProvider**(*partition*, *channel=LocalChannel(envs={}*, *script_dir=None*, *userhome='/home/docs/checkouts/readthedocs.org/user_builds/parsl/checkouts/0.9.0/)*, *nodes_per_block=1*, *cores_per_node=None*, *mem_per_node=None*, *init_blocks=1*, *min_blocks=0*, *max_blocks=10*, *parallelism=1*, *walltime='00:10:00'*, *scheduler_options="*, *worker_init="*, *cmd_timeout=10*, *exclusive=True*, *move_files=True*, *launcher=SingleNodeLauncher()*)

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.

Parameters

- **partition** (*str*) – Slurm partition to request blocks from.
- **channel** (Channel) – Channel for accessing this provider. Possible channels include *LocalChannel* (the default), *SSHChannel*, or *SSHInteractiveLoginChannel*.
- **nodes_per_block** (*int*) – Nodes to provision per block.
- **cores_per_node** (*int*) – Specify the number of cores to provision per node. If set to None, executors will assume all cores on the node are available for computation. Default is None.
- **mem_per_node** (*float*) – Specify the real memory to provision per node in GB. If set to None, no explicit request to the scheduler will be made. Default is None.
- **min_blocks** (*int*) – Minimum number of blocks to maintain.
- **max_blocks** (*int*) – Maximum number of blocks to maintain.
- **parallelism** (*float*) – Ratio of provisioned task slots to active tasks. 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., min_blocks) are used.
- **walltime** (*str*) – Walltime requested per block in HH:MM:SS.
- **scheduler_options** (*str*) – String to prepend to the #SBATCH blocks in the submit script to the scheduler.
- **worker_init** (*str*) – Command to be run before starting a worker, such as 'module load Anaconda; source activate env'.
- **exclusive** (*bool (Default = True)*) – Requests nodes which are not shared with other running jobs.
- **launcher** (*Launcher*) –

  Launcher for this provider. Possible launchers include *SingleNodeLauncher* (the default), *SrunLauncher*, or *AprunLauncher*

  move_files : Optional[Bool]: should files be moved? by default, Parsl will try to move files.

`__init__`(*partition*, *channel=LocalChannel(envs={}*, *script_dir=None*, *userhome='/home/docs/checkouts/readthedocs.org/user_builds/parsl/checkouts/0.9.0/docs')*, *nodes_per_block=1*, *cores_per_node=None*, *mem_per_node=None*, *init_blocks=1*, *min_blocks=0*, *max_blocks=10*, *parallelism=1*, *walltime='00:10:00'*, *scheduler_options=''*, *worker_init=''*, *cmd_timeout=10*, *exclusive=True*, *move_files=True*, *launcher=SingleNodeLauncher()*)

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

`_status`()

Internal: Do not call. Returns the status list for a list of job_ids

Parameters **`self`** –

Returns Status list of all jobs

Return type [status. . .]

`_write_submit_script`(*template*, *script_filename*, *job_name*, *configs*)

Generate submit script and write it to a file.

Parameters

- **`template`** (*-*) – The template string to be used for the writing submit script
- **`script_filename`** (*-*) – Name of the submit script
- **`job_name`** (*-*) – job name
- **`configs`** (*-*) – configs that get pushed into the template

Returns on success

Return type

- True

Raises

- `SchedulerMissingArgs` – If template is missing args
- `ScriptPathError` – Unable to write submit script out

`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.

`current_capacity`

Returns the currently provisioned blocks. This may need to return more information in the futures : { minsize, maxsize, current_requested }

`status`(*job_ids*)

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

Parameters **`job_ids`** (*-*) – 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

- ExecutionProviderException or its subclasses

submit(*command*, *tasks_per_node*, *job_name='parsl.auto'*)

Submit the command as a slurm job.

Parameters

- **command** (*str*) – Command to be made on the remote side.
- **tasks_per_node** (*int*) – Command invocations to be launched per node
- **job_name** (*str*) – Name for the job (must be unique).

Returns If at capacity, returns None; otherwise, a string identifier for the job

Return type None or str

Cobalt

class parsl.providers.**CobaltProvider**(*channel=LocalChannel(envs={}, script_dir=None, userhome='/home/docs/checkouts/readthedocs.org/user_builds/parsl/checkouts/0.9.), nodes_per_block=1, init_blocks=0, min_blocks=0, max_blocks=10, parallelism=1, walltime='00:10:00', account=None, queue=None, scheduler_options='', worker_init='', launcher=AprunLauncher(overrides=''), cmd_timeout=10*)

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.

Parameters

- **channel** (Channel) – Channel for accessing this provider. Possible channels include *LocalChannel* (the default), *SSHChannel*, or *SSHInteractiveLoginChannel*.
- **nodes_per_block** (*int*) – Nodes to provision per block.
- **min_blocks** (*int*) – Minimum number of blocks to maintain.
- **max_blocks** (*int*) – Maximum number of blocks to maintain.
- **walltime** (*str*) – Walltime requested per block in HH:MM:SS.
- **account** (*str*) – Account that the job will be charged against.
- **queue** (*str*) – Torque queue to request blocks from.
- **scheduler_options** (*str*) – String to prepend to the submit script to the scheduler.
- **worker_init** (*str*) – Command to be run before starting a worker, such as 'module load Anaconda; source activate env'.
- **launcher** (*Launcher*) – Launcher for this provider. Possible launchers include *AprunLauncher* (the default) or, *SingleNodeLauncher*

__init__(*channel=LocalChannel(envs={}, script_dir=None, userhome='/home/docs/checkouts/readthedocs.org/user_builds/parsl/checkouts/0.9.0/docs'), nodes_per_block=1, init_blocks=0, min_blocks=0, max_blocks=10, parallelism=1, walltime='00:10:00', account=None, queue=None, scheduler_options='', worker_init='', launcher=AprunLauncher(overrides=''), cmd_timeout=10*)

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

`_status`()

Internal: Do not call. Returns the status list for a list of job_ids

Parameters **`self`** –

Returns Status list of all jobs

Return type [status. . .]

`_write_submit_script`(*template*, *script_filename*, *job_name*, *configs*)

Generate submit script and write it to a file.

Parameters

- **`template`** (–) – The template string to be used for the writing submit script
- **`script_filename`** (–) – Name of the submit script
- **`job_name`** (–) – job name
- **`configs`** (–) – configs that get pushed into the template

Returns on success

Return type

- True

Raises

- `SchedulerMissingArgs` – If template is missing args
- `ScriptPathError` – Unable to write submit script out

`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.

`current_capacity`

Returns the currently provisioned blocks. This may need to return more information in the futures : { minsize, maxsize, current_requested }

`status`(*job_ids*)

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

Parameters **`job_ids`** (–) – 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

- ExecutionProviderException or its subclasses

`submit`(*command*, *tasks_per_node*, *job_name='parsl.auto'*)

Submits the command onto an Local Resource Manager job of 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 number of nodes are provisioned.

Parameters

- **command** (–) – (String) Commandline invocation to be made on the remote side.
- **tasks_per_node** (–) – command invocations to be launched per node

Kwargs:

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

Returns At capacity, cannot provision more - job_id: (string) Identifier for the job

Return type

- None

Condor

class parsl.providers.**CondorProvider**(*channel: parsl.channels.base.Channel = LocalChannel(envs={}, script_dir=None, userhome='/home/docs/checkouts/readthedocs.org/user_builds/parsl/checkouts/0.9.0), nodes_per_block: int = 1, cores_per_slot: Optional[int] = None, mem_per_slot: Optional[float] = None, init_blocks: int = 1, min_blocks: int = 0, max_blocks: int = 10, parallelism: float = 1, environment: Optional[Dict[str, str]] = None, project: str = ", scheduler_options: str = ", transfer_input_files: List[str] = [], walltime: str = '00:10:00', worker_init: str = ", launcher: parsl.launchers.launchers.Launcher = SingleNodeLauncher(), requirements: str = ", cmd_timeout: int = 60*)

HTCondor Execution Provider.

Parameters

- **channel** (Channel) – Channel for accessing this provider. Possible channels include *LocalChannel* (the default), *SSHChannel*, or *SSHInteractiveLoginChannel*.
- **nodes_per_block** (*int*) – Nodes to provision per block.
- **cores_per_slot** (*int*) – Specify the number of cores to provision per slot. If set to None, executors will assume all cores on the node are available for computation. Default is None.
- **mem_per_slot** (*float*) – Specify the real memory to provision per slot in GB. If set to None, no explicit request to the scheduler will be made. Default is None.
- **init_blocks** (*int*) – Number of blocks to provision at time of initialization
- **min_blocks** (*int*) – Minimum number of blocks to maintain
- **max_blocks** (*int*) – Maximum number of blocks to maintain.
- **parallelism** (*float*) – Ratio of provisioned task slots to active tasks. 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., min_blocks) are used.
- **environment** (*dict of str*) – A dictionary of environmant variable name and value pairs which will be set before running a task.

- **project** (*str*) – Project which the job will be charged against
- **scheduler_options** (*str*) – String to add specific condor attributes to the HTCondor submit script.
- **transfer_input_files** (*list (str)*) – List of strings of paths to additional files or directories to transfer to the job
- **worker_init** (*str*) – Command to be run before starting a worker.
- **requirements** (*str*) – Condor requirements.
- **launcher** (*Launcher*) – Launcher for this provider. Possible launchers include *SingleNodeLauncher* (the default),
- **cmd_timeout** (*int*) – Timeout for commands made to the scheduler in seconds

__init__(*channel: parsl.channels.base.Channel = LocalChannel(envs={}, script_dir=None, user-home='/home/docs/checkouts/readthedocs.org/user_builds/parsl/checkouts/0.9.0/docs'), nodes_per_block: int = 1, cores_per_slot: Optional[int] = None, mem_per_slot: Optional[float] = None, init_blocks: int = 1, min_blocks: int = 0, max_blocks: int = 10, parallelism: float = 1, environment: Optional[Dict[str, str]] = None, project: str = ", scheduler_options: str = ", transfer_input_files: List[str] = [], walltime: str = '00:10:00', worker_init: str = ", launcher: parsl.launchers.launchers.Launcher = SingleNodeLauncher(), requirements: str = ", cmd_timeout: int = 60*) → None
Initialize self. See help(type(self)) for accurate signature.

_status()
Update the resource dictionary with job statuses.

_write_submit_script(*template, script_filename, job_name, configs*)
Generate submit script and write it to a file.

Parameters

- **template** (*-*) – The template string to be used for the writing submit script
- **script_filename** (*-*) – Name of the submit script
- **job_name** (*-*) – job name
- **configs** (*-*) – configs that get pushed into the template

Returns on success

Return type

- True

Raises

- SchedulerMissingArgs – If template is missing args
- ScriptPathError – Unable to write submit script out

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

Parameters **job_ids** (*list of str*) – The job IDs to cancel.

Returns Each entry in the list will be True if the job is cancelled succesfully, otherwise False.

Return type list of bool

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

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

Parameters **job_ids** (`list of int`) – Identifiers of jobs for which the status will be returned.

Returns Status codes for the requested jobs.

Return type List of int

submit (*command*, *tasks_per_node*, *job_name='parsl.auto'*)
Submits the command onto an Local Resource Manager job.

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.

Parameters

- **command** (`str`) – Command to execute
- **job_name** (`str`) – Job name prefix.
- **tasks_per_node** (`int`) – command invocations to be launched per node

Returns None if at capacity and cannot provision more; otherwise the identifier for the job.

Return type None or str

Torque

class parsl.providers.**TorqueProvider** (*channel=LocalChannel(envs={}, script_dir=None, userhome='/home/docs/checkouts/readthedocs.org/user_builds/parsl/checkouts/0.9.0), account=None, queue=None, scheduler_options='', worker_init='', nodes_per_block=1, init_blocks=1, min_blocks=0, max_blocks=100, parallelism=1, launcher=AprunLauncher(overrides=''), walltime='00:20:00', cmd_timeout=120*)

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.

Parameters

- **channel** (`Channel`) – Channel for accessing this provider. Possible channels include `LocalChannel` (the default), `SSHChannel`, or `SSHInteractiveLoginChannel`.
- **account** (`str`) – Account the job will be charged against.
- **queue** (`str`) – Torque queue to request blocks from.
- **nodes_per_block** (`int`) – Nodes to provision per block.
- **init_blocks** (`int`) – Number of blocks to provision at the start of the run. Default is 1.
- **min_blocks** (`int`) – Minimum number of blocks to maintain. Default is 0.
- **max_blocks** (`int`) – Maximum number of blocks to maintain.

- **`parallelism`** (*`float`*) – Ratio of provisioned task slots to active tasks. 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., min_blocks) are used.
- **`walltime`** (*`str`*) – Walltime requested per block in HH:MM:SS.
- **`scheduler_options`** (*`str`*) – String to prepend to the #PBS blocks in the submit script to the scheduler.
- **`worker_init`** (*`str`*) – Command to be run before starting a worker, such as 'module load Anaconda; source activate env'.
- **`launcher`** (*`Launcher`*) – Launcher for this provider. Possible launchers include *`AprunLauncher`* (the default), or *`SingleNodeLauncher`*,

`__init__`(*channel=LocalChannel(envs={}, script_dir=None, userhome='/home/docs/checkouts/readthedocs.org/user_builds/parsl/checkouts/0.9.0/docs'), account=None, queue=None, scheduler_options='', worker_init='', nodes_per_block=1, init_blocks=1, min_blocks=0, max_blocks=100, parallelism=1, launcher=AprunLauncher(overrides=''), walltime='00:20:00', cmd_timeout=120*)

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

`_status`()

Internal: Do not call. Returns the status list for a list of job_ids

Parameters **`self`** –

Returns Status list of all jobs

Return type [status. . .]

`_write_submit_script`(*template, script_filename, job_name, configs*)

Generate submit script and write it to a file.

Parameters

- **`template`** (*-*) – The template string to be used for the writing submit script
- **`script_filename`** (*-*) – Name of the submit script
- **`job_name`** (*-*) – job name
- **`configs`** (*-*) – configs that get pushed into the template

Returns on success

Return type

- True

Raises

- `SchedulerMissingArgs` – If template is missing args
- `ScriptPathError` – Unable to write submit script out

`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.

`current_capacity`

Returns the currently provisioned blocks. This may need to return more information in the futures : { minsize, maxsize, current_requested }

status (*job_ids*)

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

Parameters **job_ids** (–) – 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

- ExecutionProviderException or its subclasses

submit (*command*, *tasks_per_node*, *job_name='parsl.auto'*)

Submits the command onto an Local Resource Manager job. 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 number of nodes are provisioned.

Parameters

- **command** (–) – (String) Commandline invocation to be made on the remote side.
- **tasks_per_node** (–) – command invocations to be launched per node

Kwargs:

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

Returns At capacity, cannot provision more - job_id: (string) Identifier for the job

Return type

- None

Local

class parsl.providers.**LocalProvider** (*channel=LocalChannel(envs={}, script_dir=None, user-home='/home/docs/checkouts/readthedocs.org/user_builds/parsl/checkouts/0.9.0/), nodes_per_block=1, launcher=SingleNodeLauncher(), init_blocks=4, min_blocks=0, max_blocks=10, wall-time='00:15:00', worker_init='', cmd_timeout=30, parallelism=1, move_files=None*)

Local Execution Provider

This provider is used to provide execution resources from the localhost.

Parameters

- **min_blocks** (*int*) – Minimum number of blocks to maintain.
- **max_blocks** (*int*) – Maximum number of blocks to maintain.
- **parallelism** (*float*) – Ratio of provisioned task slots to active tasks. 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., min_blocks) are used.

- **`move_files`** (*`Optional[Bool]: should files be moved? by default, Parsl will try to figure`*) – this out itself (= None). If True, then will always move. If False, will never move.
- **`worker_init`** (*`str`*) – Command to be run before starting a worker, such as 'module load Anaconda; source activate env'.

`__init__`(*channel=LocalChannel(envs={}, script_dir=None, userhome='/home/docs/checkouts/readthedocs.org/user_builds/parsl/checkouts/0.9.0/docs'), nodes_per_block=1, launcher=SingleNodeLauncher(), init_blocks=4, min_blocks=0, max_blocks=10, walltime='00:15:00', worker_init=", cmd_timeout=30, parallelism=1, move_files=None*)

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

`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.

`status`(*job_ids*)

Get the status of a list of jobs identified by their ids.

Parameters **`job_ids`** (–) – List of identifiers for the jobs

Returns

- List of status codes.

`submit`(*command, tasks_per_node, job_name='parsl.auto'*)

Submits the command onto an Local Resource Manager job. 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 nodes are provisioned.

Parameters

- **`command`** (–) – (String) Commandline invocation to be made on the remote side.
- **`tasks_per_node`** (–) – command invocations to be launched per node

Kwargs:

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

Returns At capacity, cannot provision more - job_id: (string) Identifier for the job

Return type

- None

AWS

class parsl.providers.**AWSProvider**(*image_id*, *key_name*, *init_blocks=1*, *min_blocks=0*, *max_blocks=10*, *nodes_per_block=1*, *parallelism=1*, *worker_init=''*, *instance_type='t2.small'*, *region='us-east-2'*, *spot_max_bid=0*, *key_file=None*, *profile=None*, *iam_instance_profile_arn=''*, *state_file=None*, *walltime='01:00:00'*, *linger=False*, *launcher=SingleNodeLauncher()*)

A provider for using Amazon Elastic Compute Cloud (EC2) resources.

One of 3 methods are required to authenticate: keyfile, profile or environment variables. If neither keyfile or profile are set, the following environment variables must be set: `AWS_ACCESS_KEY_ID` (the access key for your AWS account), `AWS_SECRET_ACCESS_KEY` (the secret key for your AWS account), and (optionaly) the `AWS_SESSION_TOKEN` (the session key for your AWS account).

Parameters

- **image_id** (*str*) – Identification of the Amazon Machine Image (AMI).
- **worker_init** (*str*) – String to append to the Userdata script executed in the cloudinit phase of instance initialization.
- **walltime** (*str*) – Walltime requested per block in HH:MM:SS.
- **key_file** (*str*) – Path to json file that contains 'AWSAccessKeyId' and 'AWSSecretKey'.
- **nodes_per_block** (*int*) – This is always 1 for ec2. Nodes to provision per block.
- **profile** (*str*) – Profile to be used from the standard aws config file ~/.aws/config.
- **nodes_per_block** – Nodes to provision per block. Default is 1.
- **init_blocks** (*int*) – Number of blocks to provision at the start of the run. Default is 1.
- **min_blocks** (*int*) – Minimum number of blocks to maintain. Default is 0.
- **max_blocks** (*int*) – Maximum number of blocks to maintain. Default is 10.
- **instance_type** (*str*) – EC2 instance type. Instance types comprise varying combinations of CPU, memory, . storage, and networking capacity For more information on possible instance types,. see here Default is 't2.small'.
- **region** (*str*) – Amazon Web Service (AWS) region to launch machines. Default is 'us-east-2'.
- **key_name** (*str*) – Name of the AWS private key (.pem file) that is usually generated on the console to allow SSH access to the EC2 instances. This is mostly used for debugging.
- **spot_max_bid** (*float*) – Maximum bid price (if requesting spot market machines).
- **iam_instance_profile_arn** (*str*) – Launch instance with a specific role.
- **state_file** (*str*) – Path to the state file from a previous run to re-use.
- **walltime** – Walltime requested per block in HH:MM:SS. This option is not currently honored by this provider.
- **launcher** (*Launcher*) – Launcher for this provider. Possible launchers include *SingleNodeLauncher* (the default), *SrunLauncher*, or *AprunLauncher*
- **linger** (*Bool*) – When set to True, the workers will not `halt`. The user is responsible for shutting down the node.

`__init__`(*image_id*, *key_name*, *init_blocks=1*, *min_blocks=0*, *max_blocks=10*, *nodes_per_block=1*, *parallelism=1*, *worker_init=”*, *instance_type='t2.small'*, *region='us-east-2'*, *spot_max_bid=0*, *key_file=None*, *profile=None*, *iam_instance_profile_arn=”*, *state_file=None*, *walltime='01:00:00'*, *linger=False*, *launcher=SingleNodeLauncher()*)
Initialize self. See help(type(self)) for accurate signature.

`cancel`(*job_ids*)
Cancel the jobs specified by a list of job ids.

Parameters **`job_ids`** (*`list of str`*) – List of of job identifiers

Returns Each entry in the list will contain False if the operation fails. Otherwise, the entry will be True.

Return type list of bool

`create_session`()
Create a session.

First we look in self.key_file for a path to a json file with the credentials. The key file should have 'AWSAccessKeyId' and 'AWSSecretKey'.

Next we look at self.profile for a profile name and try to use the Session call to automatically pick up the keys for the profile from the user default keys file ~/.aws/config.

Finally, boto3 will look for the keys in environment 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

We create a VPC with CIDR 10.0.0.0/16, which provides up to 64,000 instances.

We attach a subnet for each availability zone within the region specified in the config. We give each subnet an ip range like 10.0.X.0/20, which is large enough for approx. 4000 instances.

Security groups are configured in function security_group.

`current_capacity`
Returns the current blocksize.

`read_state_file`(*state_file*)
Read the state file, if it exists.

If this script has been run previously, resource IDs will have been written to a state file. On starting a run, a state file will be looked for before creating new infrastructure. Information on VPCs, security groups, and subnets are saved, as well as running instances and their states.

AWS has a maximum number of VPCs per region per account, so we do not want to clutter users' AWS accounts with security groups and VPCs that will be used only once.

`security_group`(*vpc*)
Create and configure a new security group.

Allows all ICMP in, all TCP and UDP in within VPC.

This security group is very open. It allows all incoming ping requests on all ports. It also allows all outgoing traffic on all ports. This can be limited by changing the allowed port ranges.

Parameters **`vpc`** (*`VPC instance`*) – VPC in which to set up security group.

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

Parameters **job_ids** (*list of str*) – Identifiers for the jobs.

Returns The status codes of the requsted jobs.

Return type list of int

submit (*command='sleep 1'*, *tasks_per_node=1*, *job_name='parsl.auto'*)
Submit the command onto a freshly instantiated AWS EC2 instance.

Submit returns an ID that corresponds to the task that was just submitted.

Parameters

- **command** (*str*) – Command to be invoked on the remote side.
- **tasks_per_node** (*int (default=1)*) – Number of command invocations to be launched per node
- **job_name** (*str*) – Prefix for the job name.

Returns If at capacity, None will be returned. Otherwise, the job identifier will be returned.

Return type None or str

write_state_file ()
Save information that must persist to a file.

We do not want to create a new VPC and new identical security groups, so we save information about them in a file between runs.

GridEngine

class parsl.providers.**GridEngineProvider** (*channel=LocalChannel(envs={}, script_dir=None, userhome='/home/docs/checkouts/readthedocs.org/user_builds/parsl/checkout), nodes_per_block=1, init_blocks=1, min_blocks=0, max_blocks=10, parallelism=1, walltime='00:10:00', scheduler_options='', worker_init='', launcher=SingleNodeLauncher()*)
A provider for the Grid Engine scheduler.

Parameters

- **channel** (*Channel*) – Channel for accessing this provider. Possible channels include *LocalChannel* (the default), *SSHChannel*, or *SSHInteractiveLoginChannel*.
- **nodes_per_block** (*int*) – Nodes to provision per block.
- **min_blocks** (*int*) – Minimum number of blocks to maintain.
- **max_blocks** (*int*) – Maximum number of blocks to maintain.
- **parallelism** (*float*) – Ratio of provisioned task slots to active tasks. 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., min_blocks) are used.
- **walltime** (*str*) – Walltime requested per block in HH:MM:SS.

- **`scheduler_options`** (*`str`*) – String to prepend to the #$$ blocks in the submit script to the scheduler.
- **`worker_init`** (*`str`*) – Command to be run before starting a worker, such as 'module load Anaconda; source activate env'.
- **`launcher`** (*`Launcher`*) – Launcher for this provider. Possible launchers include *`SingleNodeLauncher`* (the default),

`__init__`(*channel=LocalChannel(envs={}, script_dir=None, userhome='/home/docs/checkouts/readthedocs.org/user_builds/parsl/checkouts/0.9.0/docs'), nodes_per_block=1, init_blocks=1, min_blocks=0, max_blocks=10, parallelism=1, walltime='00:10:00', scheduler_options='', worker_init='', launcher=SingleNodeLauncher()*)

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

`cancel`(*job_ids*)

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

Parameters **`job_ids`** (*–*) – A list of job identifiers

Returns

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

Raises

- ExecutionProviderException or its subclasses

`current_capacity`

Returns the currently provisioned blocks. This may need to return more information in the futures : { minsize, maxsize, current_requested }

`status`(*job_ids*)

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

Parameters **`job_ids`** (*–*) – 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

- ExecutionProviderException or its subclasses

`submit`(*command, tasks_per_node, 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 :

- command (str) : The bash command string to be executed.
- tasks_per_node (int) : command invocations to be launched per node

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

- ExecutionProviderException or its subclasses

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 parsl.channels.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.

```
                      +------------------
                      |
cmd, wtime    ------->|  execute_wait
(ec, stdout, stderr)<-|---+
                      |
cmd, wtime    ------->|  execute_no_wait
(ec, stdout, stderr)<-|---+
                      |
src, dst_dir  ------->|  push_file
   dst_path  <--------|----+
                      |
dst_script_dir <------|  script_dir
                      |
                      +-------------------
```

close()

Closes the channel. Clean out any auth credentials.

Parameters **None** –

Returns Bool

execute_no_wait(*cmd*, *walltime*, *envs={}*, **args*, ***kwargs*)

Execute asynchronousely without waiting for exitcode

Parameters

- **cmd** (*–*) – Command string to execute over the channel
- **walltime** (*–*) – Timeout in seconds

KWargs:

- envs (dict) : Environment variables to push to the remote side

Returns

- the type of return value is channel specific

execute_wait(*cmd*, *walltime=None*, *envs={}*, **args*, ***kwargs*)

Executes the cmd, with a defined walltime.

Parameters

- **cmd** (*–*) – Command string to execute over the channel
- **walltime** (*–*) – Timeout in seconds, optional

KWargs:

- envs (Dict[str, str]) : Environment variables to push to the remote side

Returns

- (exit_code, stdout, stderr) (int, optional string, optional string) If the exit code is a failure code, the stdout and stderr return values may be None.

push_file(*source*, *dest_dir*)
Channel will take care of moving the file from source to the destination directory

Parameters

- **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.

Parameters **None** (*-*) –

Returns

- Channel script dir

LocalChannel

class parsl.channels.**LocalChannel**(*userhome='.'*, *envs={}*, *script_dir=None*, ***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={}*, *script_dir=None*, ***kwargs*)
Initialize the local channel. script_dir 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
- script_dir (string): Directory to place scripts

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*, *envs={}*)
Synchronously execute a commandline string on the shell.

Parameters

- **cmd** (*-*) – Commandline string to execute
- **walltime** (*-*) – walltime in seconds, this is not really used now.

Returns a tuple containing:

- pid : process id
- proc : a subprocess.Popen object

Raises `None.`

`execute_wait`(*cmd*, *walltime=None*, *envs={}*)
Synchronously execute a commandline string on the shell.

Parameters

- **`cmd`** (–) – Commandline string to execute
- **`walltime`** (–) – walltime in seconds, this is not really used now.

Kwargs:

- envs (dict) : Dictionary of env variables. This will be used to override the envs set at channel initialization.

Returns Return code from the execution, -1 on fail - stdout : stdout string - stderr : stderr string

Return type

- retcode

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.

Parameters

- **`source`** (–) – Path to the source file
- **`dest_dir`** (–) – Path to the directory to which the files is to be copied

Returns Absolute path of the destination file

Return type

- destination_path (String)

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.

Parameters **`None`** (–) –

Returns

- Channel script dir

SSHChannel

class `parsl.channels.`**`SSHChannel`**(*hostname*, *username=None*, *password=None*, *script_dir=None*, *envs=None*, *gssapi_auth=False*, *skip_auth=False*, *port=22*, ***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*, *script_dir=None*, *envs=None*, *gss-api_auth=False*, *skip_auth=False*, *port=22*, ***kwargs*)

Initialize a persistent connection to the remote system. We should know at this point whether ssh connectivity is possible

Parameters **`hostname`** (*–*) – Hostname

KWargs:

- username (string) : Username on remote system
- password (string) : Password for remote system
- port : The port designated for the ssh connection. Default is 22.
- script_dir (string) : Full path to a script dir where generated scripts could be sent to.
- envs (dict) : A dictionary of environment variables to be set when executing commands

Raises:

`close`()

Closes the channel. Clean out any auth credentials.

Parameters **`None`** –

Returns Bool

`execute_no_wait`(*cmd*, *walltime=2*, *envs={}*)

Execute asynchronousely without waiting for exitcode

Parameters

- **`cmd`** (*–*) – Commandline string to be executed on the remote side
- **`walltime`** (*–*) – timeout to exec_command

KWargs:

- envs (dict): A dictionary of env variables

Returns

- None, stdout (readable stream), stderr (readable stream)

Raises

- ChannelExecFailed (reason)

`execute_wait`(*cmd*, *walltime=2*, *envs={}*)

Synchronously execute a commandline string on the shell.

Parameters

- **`cmd`** (*–*) – Commandline string to execute
- **`walltime`** (*–*) – walltime in seconds

Kwargs:

- envs (dict) : Dictionary of env variables

Returns Return code from the execution, -1 on fail - stdout : stdout string - stderr : stderr string

Return type

- retcode

Raises: None.

`pull_file`(*remote_source*, *local_dir*)
Transport file on the remote side to a local directory

Parameters

- **`remote_source`** (–) – remote_source
- **`local_dir`** (–) – Local directory to copy to

Returns Local path to file

Return type

- str

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

Parameters

- **`local_source`** (–) – Path
- **`remote_dir`** (–) – Remote path

Returns Path to copied file on remote machine

Return type

- str

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.

`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.

Parameters **`None`** (–) –

Returns

- Channel script dir

SSHILChannel

class parsl.channels.**SSHInteractiveLoginChannel**(*hostname*, *username=None*, *password=None*, *script_dir=None*, *envs=None*, ***kwargs*)

SSH persistent channel. This enables remote execution on sites accessible via ssh. This channel supports interactive login and is appropriate when keys are not set up.

__init__(*hostname*, *username=None*, *password=None*, *script_dir=None*, *envs=None*, ***kwargs*)

Initialize a persistent connection to the remote system. We should know at this point whether ssh connectivity is possible

Parameters **hostname** (–) – Hostname

KWargs:

- username (string) : Username on remote system
- password (string) : Password for remote system
- script_dir (string) : Full path to a script dir where generated scripts could be sent to.
- envs (dict) : A dictionary of env variables to be set when executing commands

Raises:

close()

Closes the channel. Clean out any auth credentials.

Parameters **None** –

Returns Bool

execute_no_wait(*cmd*, *walltime=2*, *envs={}*)

Execute asynchronousely without waiting for exitcode

Parameters

- **cmd** (–) – Commandline string to be executed on the remote side
- **walltime** (–) – timeout to exec_command

KWargs:

- envs (dict): A dictionary of env variables

Returns

- None, stdout (readable stream), stderr (readable stream)

Raises

- ChannelExecFailed (reason)

execute_wait(*cmd*, *walltime=2*, *envs={}*)

Synchronously execute a commandline string on the shell.

Parameters

- **cmd** (–) – Commandline string to execute
- **walltime** (–) – walltime in seconds

Kwargs:

- envs (dict) : Dictionary of env variables

Returns Return code from the execution, -1 on fail - stdout : stdout string - stderr : stderr string

Return type

- retcode

Raises: None.

`pull_file`(*remote_source*, *local_dir*)
Transport file on the remote side to a local directory

Parameters

- **`remote_source`** (–) – remote_source
- **`local_dir`** (–) – Local directory to copy to

Returns Local path to file

Return type

- str

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

Parameters

- **`local_source`** (–) – Path
- **`remote_dir`** (–) – Remote path

Returns Path to copied file on remote machine

Return type

- str

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.

`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.

Parameters **`None`** (–) –

Returns

- Channel script dir

6.6.11 Launchers

Launchers are basically wrappers for user submitted scripts as they are submitted to a specific execution resource.

SimpleLauncher

class parsl.launchers.**SimpleLauncher**

Does no wrapping. Just returns the command as-is

SingleNodeLauncher

class parsl.launchers.**SingleNodeLauncher**

Worker launcher that wraps the user's command with the framework to launch multiple command invocations in parallel. This wrapper sets the bash env variable CORES to the number of cores on the machine. By setting task_blocks to an integer or to a bash expression the number of invocations of the command to be launched can be controlled.

AprunLauncher

class parsl.launchers.**AprunLauncher**(*overrides=”*)

Worker launcher that wraps the user's command with the Aprun launch framework to launch multiple cmd invocations in parallel on a single job allocation

SrunLauncher

class parsl.launchers.**SrunLauncher**(*overrides=”*)

Worker launcher that wraps the user's command with the SRUN launch framework to launch multiple cmd invocations in parallel on a single job allocation.

SrunMPILauncher

class parsl.launchers.**SrunMPILauncher**(*overrides=”*)

Launches as many workers as MPI tasks to be executed concurrently within a block.

Use this launcher instead of SrunLauncher if each block will execute multiple MPI applications at the same time. Workers should be launched with independent Srun calls so as to setup the environment for MPI application launch.

6.6.12 Flow Control

This section deals with functionality related to controlling the flow of tasks to various executors.

FlowControl

class parsl.dataflow.flow_control.**FlowControl**(*dfk*, **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*, **args*, *threshold=2*, *interval=2*)
FlowNoControl implements similar interfaces as FlowControl.

Null handlers are used so as to mimic the FlowControl class.

__init__(*dfk*, **args*, *threshold=2*, *interval=2*)
Initialize the flowcontrol object. This does nothing.

Parameters **dfk** (–) – DFK object to track parsl progress

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*, *name=None*)

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*, *name=None*)

Initialize the flowcontrol object We start the timer thread here

Parameters **dfk** (*-*) – DFK object to track parsl progress

KWargs:

- threshold (int) : Tasks after which the callback is triggered
- interval (int) : seconds after which timer expires
- name (str) : a base name to use when naming the started thread

__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 executors 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 an executor. An executor 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.

```
              |<--min_blocks     |<-init_blocks              max_blocks-->|
              +----------------------------------------------------------+
              |  +--------block----------+       +--------block--------+ |
executor = |  | task           task     | ...   |    task      task    | |
```

(continues on next page)

(continued from previous page)

```
            |   +------------------------+       +----------------------+ |
            +------------------------------------------------------------+
```

The relevant specification options are:

1. min_blocks: Minimum number of blocks to maintain
2. init_blocks: number of blocks to provision at initialization of workflow
3. max_blocks: Maximum number of blocks that can be active due to one workflow

```
slots = current_capacity * tasks_per_node * nodes_per_block

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 =  min_blocks           { if active_tasks = 0
          max(min_blocks, 1)   {  else
```

When p = 1, => compute with the most resources. one task is stacked per slot.

```
blocks = min ( max_blocks,
         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 task_blocks=2 Consider the following example: min_blocks = 0 init_blocks = 0 max_blocks = 4 tasks_per_node = 2 nodes_per_block = 1

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)

unset_logging()
Mute newly added handlers to the root level, right after calling executor.status

6.6.13 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)
   |     |     |                                (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.

Parameters **dfk** (–) – 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 None result is possible and could be confusing. This seems like a reasonable option without relying on a cache_miss exception.

Parameters **task** (–) – task from the dfk.tasks table

Returns

- present (Bool): Is this present in the memo_lookup_table
- Result (Py Obj): Result of the function if present in table

Return type Tuple of the following

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.

Parameters **hashsum** (–) – The same hashes used to uniquely identify apps+inputs

Returns

- Lookup result

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.

Parameters **task** (–) – Task dictionary from dfk.tasks

Returns A unique hash string

Return type

- hash (str)

update_memo (*task_id*, *task*, *r*)
Updates the memoization lookup table with the result from a task.

Parameters

- **task_id** (–) – Integer task id
- **task** (–) – A task dict from dfk.tasks
- **r** (–) – Result future

A warning is issued when a hash collision occurs during the update. This is not likely.

6.7 Packaging

Currently packaging is managed by @annawoodard and @yadudoc.

Steps to release

1. Update the version number in `parsl/parsl/version.py`
2. Check the following files to confirm new release information * `parsl/setup.py` * `requirements.txt` * `parsl/docs/devguide/changelog.rst` * `parsl/README.rst`

4. Commit and push the changes to github 3. Run the `tag_and_release.sh` script. This script will verify that

version number matches the version specified.

```
./tag_and_release.sh <VERSION_FOR_TAG>
```

Here are the steps that is taken by the `tag_and_release.sh` script:

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

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

# Make sure to have twine installed
pip3 install twine

# Create a source distribution
python3 setup.py sdist

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

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

CHAPTER 7

Indices and tables

- genindex
- modindex
- search

Python Module Index

Index

Symbols