
Parsl Documentation

Release 1.1.0

The Parsl Team

Apr 26, 2021

CONTENTS

1	Quickstart	3
1.1	Installation	3
1.2	Getting started	4
1.3	Tutorial	4
1.4	Usage Tracking	4
1.5	For Developers	5
2	Parsl tutorial	7
2.1	Configuring Parsl	7
2.2	Python Apps	8
2.3	Bash Apps	8
2.4	Passing data	9
2.5	AppFutures	9
2.6	DataFutures	10
2.7	Files	11
2.8	Remote Files	11
2.9	Sequential workflow	12
2.10	Parallel workflow	12
2.11	Parallel dataflow	13
2.12	Monte Carlo workflow	14
2.13	Local execution with threads	15
2.14	Local execution with pilot jobs	15
3	User guide	17
3.1	Overview	17
3.2	Apps	23
3.3	Futures	26
3.4	Passing Python objects	29
3.5	Staging data files	29
3.6	Execution	35
3.7	Error handling	42
3.8	Memoization and checkpointing	43
3.9	Configuration	48
3.10	Monitoring	71
3.11	Example parallel patterns	77
3.12	Structuring Parsl programs	81
3.13	Join Apps	82
3.14	Performance and Scalability	85
3.15	Usage statistics collection	87

4	FAQ	89
4.1	How can I debug a Parsl script?	89
4.2	How can I view outputs and errors from apps?	89
4.3	How can I make an App dependent on multiple inputs?	89
4.4	Can I pass any Python object between apps?	90
4.5	How do I specify where apps should be run?	90
4.6	Workers do not connect back to Parsl	90
4.7	parsl.dataflow.error.ConfigurationError	91
4.8	Remote execution fails with SystemError(unknown opcode)	91
4.9	Parsl complains about missing packages	92
4.10	zmq.error.ZMQError: Invalid argument	92
4.11	How do I run code that uses Python2.X?	92
4.12	Parsl hangs	93
4.13	How can I start a Jupyter notebook over SSH?	93
4.14	How can I sync my conda environment and Jupyter environment?	94
4.15	Addressing SerializationError	94
4.16	How do I cite Parsl?	94
5	API Reference guide	97
5.1	Core	97
5.2	Configuration	101
5.3	Channels	105
5.4	Data management	110
5.5	Executors	118
5.6	Launchers	134
5.7	Providers	138
5.8	Exceptions	157
5.9	Internal	164
6	Developer documentation	173
6.1	Contributing	173
6.2	Changelog	173
6.3	Libsubmit Changelog	191
6.4	Swift vs Parsl	192
6.5	Roadmap	195
6.6	Packaging	198
6.7	Doc Docs	199
7	Indices and tables	201
	Index	203

Parsl is a flexible and scalable parallel programming library for Python. Parsl augments Python with simple constructs for encoding parallelism. Developers annotate Python functions to specify opportunities for concurrent execution. These annotated functions, called `apps`, may represent pure Python functions or calls to external applications. Parsl further allows invocations of these apps, called `tasks`, to be connected by shared input/output data (e.g., Python objects or files) via which Parsl constructs a dynamic dependency graph of tasks to manage concurrent task execution where possible.

Parsl includes an extensible and scalable runtime that allows it to efficiently execute Parsl programs on one or many processors. Parsl programs are portable, enabling them to be easily moved between different execution resources: from laptops to supercomputers. When executing a Parsl program, developers must define (or import) a Python configuration object that outlines where and how to execute tasks. Parsl supports various target resources including clouds (e.g., Amazon Web Services and Google Cloud), clusters (e.g., using Slurm, Torque/PBS, HTCondor, Cobalt), and container orchestration systems (e.g., Kubernetes). Parsl scripts can scale from several cores on a single computer through to hundreds of thousands of cores across many thousands of nodes on a supercomputer.

Parsl can be used to implement various parallel computing paradigms:

- Concurrent execution of tasks in a bag-of-tasks program.
- Procedural workflows in which tasks are executed following control logic.
- Parallel dataflow in which tasks are executed when their data dependencies are met.
- Many-task applications in which many computing resources are used to perform various computational tasks.
- Dynamic workflows in which the workflow is dynamically determined during execution.
- Interactive parallel programming through notebooks or interactive.

QUICKSTART

To try Parsl now (without installing any code locally), experiment with our [hosted tutorial notebooks on Binder](#).

1.1 Installation

Parsl is available on [PyPI](#) and [conda-forge](#).

Parsl requires Python3.5+ and has been tested on Linux and macOS.

1.1.1 Installation using Pip

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

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`

1.1.2 Installation using Conda

1. Create and activate a new conda environment:

```
$ 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
```

The conda documentation provides [instructions](#) for installing conda on macOS and Linux.

1.2 Getting started

Parsl enables concurrent execution of Python functions (*python_app*) or external applications (*bash_app*). Developers must first annotate functions with Parsl decorators. When these functions are invoked, Parsl will manage the asynchronous execution of the function on specified resources. The result of a call to a Parsl app is an *AppFuture*.

The following example shows how to write a simple Parsl program with hello world Python and Bash apps.

```
import parsl
from parsl import python_app, bash_app

parsl.load()

@python_app
def hello_python (message):
    return 'Hello %s' % message

@bash_app
def hello_bash(message, stdout='hello-stdout'):
    return 'echo "Hello %s"' % message

# invoke the Python app and print the result
print(hello_python('World (Python)').result())

# invoke the Bash app and read the result from a file
hello_bash('World (Bash)').result()

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

1.3 Tutorial

The best way to learn more about Parsl is by reviewing the Parsl tutorials. There are several options for following the tutorial:

1. Use [Binder](#) to follow the tutorial online without installing or writing any code locally.
2. Clone the [Parsl tutorial repository](#) using a local Parsl installation.
3. Read through the online [tutorial documentation](#).

1.4 Usage Tracking

To help support the Parsl project, we ask that users opt-in to anonymized usage tracking whenever possible. Usage tracking allows us to measure usage, identify bugs, and improve usability, reliability, and performance. Only aggregate usage statistics will be used for reporting purposes.

As an NSF-funded project, our ability to track usage metrics is important for continued funding.

You can opt-in by setting `PARSL_TRACKING=true` in your 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](#)*.

1.5 For Developers

Parsl is an open source community that encourages contributions from users and developers. A guide for [contributing to Parsl](#) is available in the [Parsl GitHub repository](#).

The following instructions outline how to set up Parsl from source.

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_package>...]
```

3. Use Parsl!

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 concatenate these files using `cat`. We pass in the list of hello files (`inputs`) and concatenate the text into a new file named `all_hellos.txt` (`outputs`). As we describe below we use Parsl File objects to abstract file locations in the event the `cat` app is executed on a different computer.

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

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

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

(continues on next page)

(continued from previous page)

```
time.sleep(5)
return 'Hello World!'

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=[File(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 access to a file irrespective of where the app is executed. When referencing a Parsl file in an app (by calling `filepath`), Parsl translates the path to the file's location relative to the file system on which the app is executing.

```
[ ]: 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

The Parsl file abstraction can also represent remotely accessible files. In this case, you can instantiate a file object using the remote location of the file. Parsl will implicitly 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
```

(continues on next page)

(continued from previous page)

```
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=[File(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):
    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 )) &> {}".format(outputs[0])

# App that concatenates input files into a single output file
@bash_app
def concat(inputs=[], outputs=[]):
    return "cat {0} > {1}".format(" ".join([i.filepath for i in 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=[File(os.path.join(os.getcwd(), 'random-{}.
→txt'.format(i))]))))

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

(continues on next page)

(continued from previous page)

```
# 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

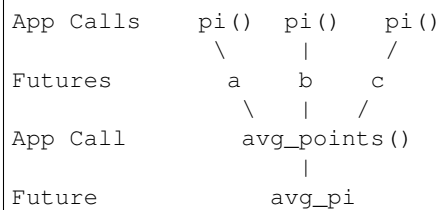
Many scientific applications use the [Monte Carlo method](#) to compute results.

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 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)
```

(continues on next page)

(continued from previous page)

```
# 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(
```

(continues on next page)

(continued from previous page)

```

        label="htex_Local",
        worker_debug=True,
        cores_per_worker=1,
        provider=LocalProvider(
            channel=LocalChannel(),
            init_blocks=1,
            max_blocks=1,
        ),
    )
],
strategy=None,
)

```

```

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

```

```

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

      @bash_app
      def concat(inputs=[], outputs=[]):
          return "cat {} > {}".format(" ".join(i.filepath for i in 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=[File(os.path.join(os.getcwd(), 'random-%s.
      ↪txt' % i)])))

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

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

      print (total.result())

```

3.1 Overview

Parsl is designed to enable straightforward parallelism and orchestration of asynchronous tasks into dataflow-based workflows, in Python. Parsl manages the concurrent execution of these tasks across various computation resources, from laptops to supercomputers, scheduling each task only when its dependencies (e.g., input data dependencies) are met.

Developing a Parsl program is a two-step process:

1. Define Parsl apps by annotating Python functions to indicate that they can be executed concurrently.
2. Use standard Python code to invoke Parsl apps, creating asynchronous tasks and adhering to dependencies defined between apps.

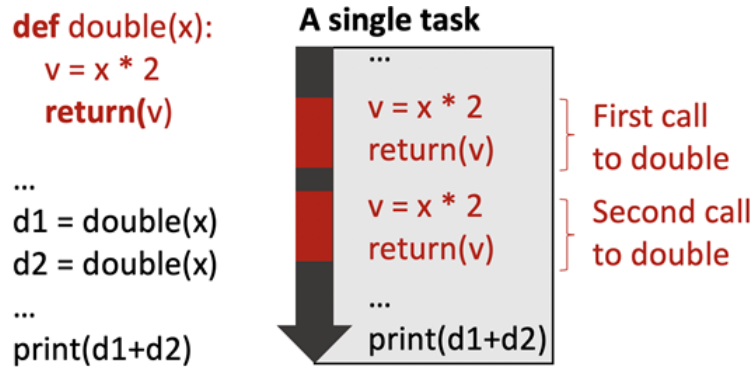
We aim in this section to provide a mental model of how Parsl programs behave. We discuss how Parsl programs create concurrent tasks, how tasks communicate, and the nature of the environment on which Parsl programs can perform operations. In each case, we compare and contrast the behavior of Python programs that use Parsl constructs with those of conventional Python programs.

Note: The behavior of a Parsl program can vary in minor respects depending on the Executor used (see [Execution](#)). We focus here on the behavior seen when using the recommended [HighThroughputExecutor](#) (HTEX).

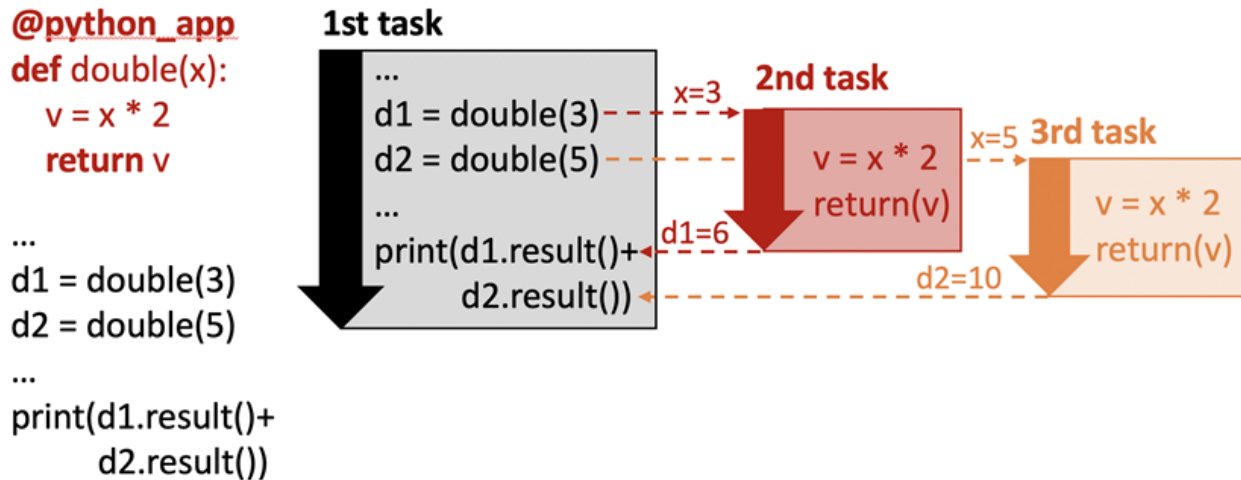
3.1.1 Parsl and Concurrency

Any call to a Parsl app creates a new task that executes concurrently with the main program and any other task(s) that are currently executing. Different tasks may execute on the same nodes or on different nodes, and on the same or different computers.

The Parsl execution model thus differs from the Python native execution model, which is inherently sequential. A Python program that does not contain Parsl constructs, or make use of other concurrency mechanisms, executes statements one at a time, in the order that they appear in the program. This behavior is illustrated in the following figure, which shows a Python program on the left and, on the right, the statements executed over time when that program is run, from top to bottom. Each time that the program calls a function, control passes from the main program (in black) to the function (in red). Execution of the main program resumes only after the function returns.



In contrast, the Parsl execution model is inherently concurrent. Whenever a program calls an app, a separate thread of execution is created, and the main program continues without pausing. Thus in the example shown in the figure below. There is initially a single task: the main program (black). The first call to `double` creates a second task (red) and the second call to `double` creates a third task (orange). The second and third task terminate as the function that they execute returns. (The dashed lines represent the start and finish of the tasks). The calling program will only block (wait) when it is explicitly told to do so (in this case by calling `result()`)



Note: Note: We talk here about concurrency rather than parallelism for a reason. Two activities are concurrent if they can execute at the same time. Two activities occur in parallel if they do run at the same time. If a Parsl program creates more tasks than there are available processors, not all concurrent activities may run in parallel.

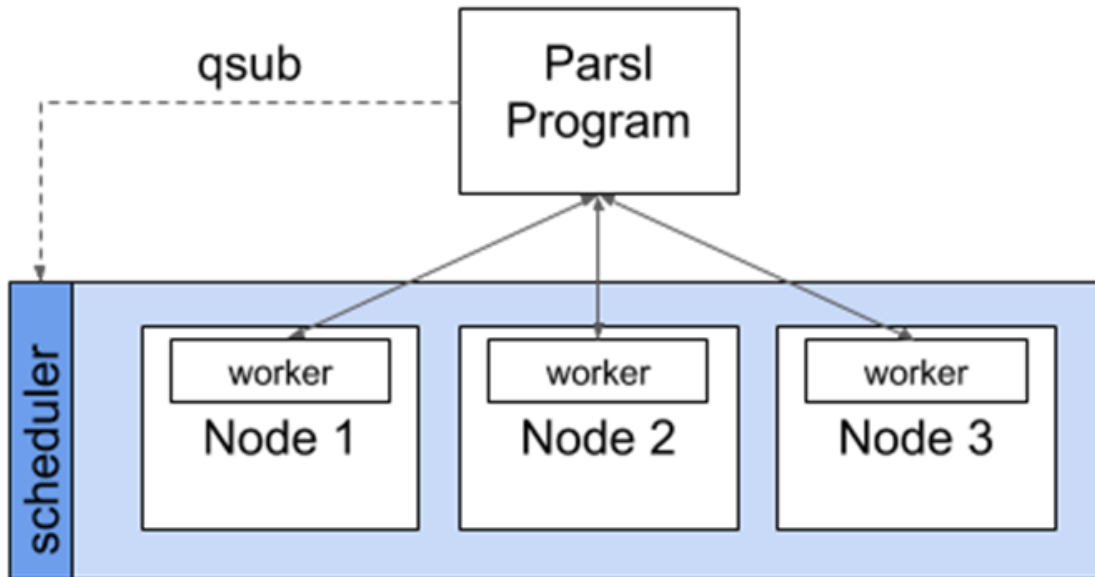
3.1.2 Parsl and Execution

We have now seen that Parsl tasks are executed concurrently alongside the main Python program and other Parsl tasks. We now turn to the question of how and where are those tasks executed. Given the range of computers on which parallel programs may be executed, Parsl allows tasks to be executed using different executors (`parsl.executors`). Executors are responsible for taking a queue of tasks and executing them on local or remote resources.

We briefly describe two of Parsl's most commonly used executors. Other executors are described in [Execution](#).

The *HighThroughputExecutor* (HTEX) implements a *pilot job model* that enables fine-grain task execution using across one or more provisioned nodes. HTEX can be used on a single node (e.g., a laptop) and will make use of multiple processes for concurrent execution. As shown in the following figure, HTEX uses Parsl's provider abstraction (`parsl.providers`) to communicate with a resource manager (e.g., batch scheduler or cloud API) to provision a set of nodes (e.g., Parsl will use Slurm's `qsub` command to request nodes on a Slurm cluster) for the duration of

execution. HTEX deploys a lightweight worker agent on the nodes which subsequently connects back to the main Parsl process. Parsl tasks are then sent from the main program to the connected workers for execution and the results are sent back via the same mechanism. This approach has a number of advantages over other methods: it avoids long job scheduler queue delays by acquiring one set of resources for the entire program and it allows for scheduling of many tasks on individual nodes.



The *ThreadPoolExecutor* allows tasks to be executed on a pool of locally accessible threads. As execution occurs on the same computer, on a pool of threads forked from the main program, the tasks share memory with one another (this is discussed further in the following sections).

3.1.3 Parsl and Communication

Parsl tasks typically need to communicate in order to perform useful work. Parsl provides for two forms of communication: by parameter passing and by file passing. As described in the next section, Parsl programs may also communicate by interacting with shared filesystems and services its environment.

Parameter Passing

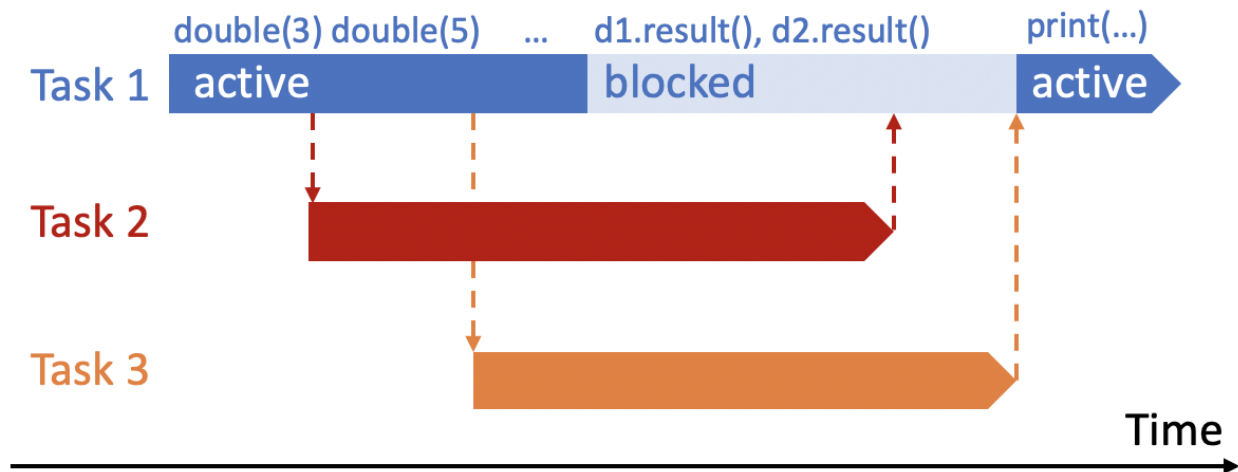
The figure above illustrates communication via parameter passing. The call `double(3)` to the app `double` in the main program creates a new task and passes the parameter value, 3, to that new task. When the task completes execution, its return value, 6, is returned to the main program. Similarly, the second task is passed the value 5 and returns the value 10. In this case, the parameters passed are simple primitive types (i.e., integers); however, complex objects (e.g., Numpy Arrays, Pandas DataFrames, custom objects) can also be passed to/from tasks.

File Passing

Parsl supports communication via files in both Bash apps and Python apps. Files may be used in place of parameter passing for many reasons, such as for apps are designed to support files, when data to be exchanged are large, or when data cannot be easily serialized into Python objects. As Parsl tasks may be executed on remote nodes, without shared file systems, Parsl offers a Parsl `parsl.data_provider.files.File` construct for location-independent reference to files. Parsl will translate file objects to worker-accessible paths when executing dependent apps. Parsl is also able to transfer files in, out, and between Parsl apps using one of several methods (e.g., FTP, HTTP(S), Globus and rsync). To accommodate the asynchronous nature of file transfer, Parsl treats data movement like a Parsl app, adding a dependency to the execution graph and waiting for transfers to complete before executing dependent apps. More information is provided in [Passing Python objects](#).

Futures

Communication via parameter and file passing also serves a second purpose, namely synchronization. As we discuss in more detail in [Futures](#), a call to an app returns a special object called a future that has a special unassigned state until such time as the app returns, at which time it takes the return value. (In the example program, two futures are thus created, d1 and d2.) The AppFuture function `result()` blocks until the future to which it is applied takes a value. Thus the print statement in the main program blocks until both child tasks created by the calls to the double app return. The following figure captures this behavior, with time going from left to right rather than top to bottom as in the preceding figure. Task 1 is initially active as it starts Tasks 2 and 3, then blocks as a result of calls to `d1.result()` and `d2.result()`, and when those values are available, is active again.



3.1.4 The Parsl Environment

Regular Python and Parsl-enhanced Python differ in terms of the environment in which code executes. We use the term *environment* here to refer to the variables and modules (the *memory environment*), the file system(s) (the *file system environment*), and the services (the *service environment*) that are accessible to a function.

An important question when it comes to understanding the behavior of Parsl programs is the environment in which this new task executes: does it have the same or different memory, file system, or service environment as its parent task or any other task? The answer, depends on the executor used, and (in the case of the file system environment) where the task executes. Below we describe behavior for the most commonly used `HighThroughputExecutor` which is representative of all Parsl executors except the `ThreadPoolExecutor`.

Memory environment

In Python, the variables and modules that are accessible to a function are defined by Python scoping rules, by which a function has access to both variables defined within the function (*local* variables) and those defined outside the function (*global* variables). Thus in the following code, the print statement in the `print_answer` function accesses the global variable “`answer`”, and we see as output “the answer is 42.”

```
answer = 42

def print_answer():
    print('the answer is', answer)

print_answer()
```

In Parsl (except when using the *ThreadPoolExecutor*) a Parsl app is executed in a distinct environment that only has access to local variables associated with the app function. Thus, if the program above is executed with say the *HighThroughputExecutor*, will print “the answer is 0” rather than “the answer is 42,” because the print statement in `provide_answer` does not have access to the global variable that has been assigned the value 42. The program will run without errors when using the *ThreadPoolExecutor*.

Similarly, the same scoping rules apply to import statements, and thus the following program will run without errors with the *ThreadPoolExecutor*, but raise errors when run with any other executor, because the return statement in `ambiguous_double` refers to a variable (`factor`) and a module (`random`) that are not known to the function.

```
import random
factor = 5

@python_app
def ambiguous_double(x):
    return x * random.random() * factor

print(ambiguous_double(42))
```

To allow this program to run correctly with all Parsl executors, the `random` library must be imported within the app, and the `factor` variable must be passed as an argument, as follows.

```
import random
factor = 5

@python_app
def good_double(factor, x):
    import random
    return x * random.random() * factor

print(good_double(factor, 42))
```

File system environment

In a regular Python program the environment that is accessible to a Python program also includes the file system(s) of the computer on which it is executing. Thus in the following code, a value written to a file “`answer.txt`” in the current directory can be retrieved by reading the same file, and the print statement outputs “the answer is 42.”

```
def print_answer_file():
    with open('answer.txt', 'r') as f:
        print('the answer is', f.read())
```

(continues on next page)

(continued from previous page)

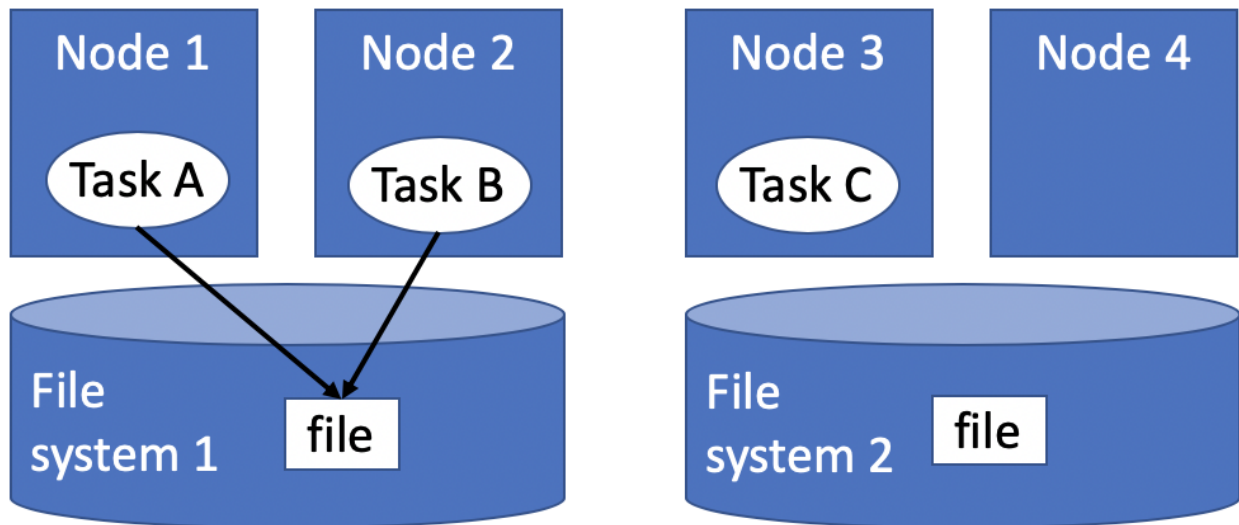
```

with open('answer.txt','w') as f:
    f.write('42')
    f.close()

print_answer_file()

```

The question of which file system environment is accessible to a Parsl app depends on where the app executes. If two tasks run on nodes that share a file system, then those tasks (e.g., tasks A and B in the figure below, but not task C) share a file system environment. Thus the program above will output “the answer is 42” if the parent task and the child task run on nodes 1 and 2, but not if they run on nodes 2 and 3.



Service Environment

We use the term service environment to refer to network services that may be accessible to a Parsl program, such as a Redis server or Globus data management service. These services are accessible to any task.

Environment Summary

As we summarize in the table, if tasks execute with the *ThreadPoolExecutor*, they share the memory and file system environment of the parent task. If they execute with any other executor, they have a separate memory environment, and may or may not share their file system environment with other tasks, depending on where they are placed. All tasks typically have access to the same network services.

	Share memory environment with parent/other tasks	Share file system environment with parent	Share file system environment with other tasks	Share service environment with other tasks
Python without Parsl	Yes	Yes	N/A	N/A
Parsl Thread-PoolExecutor	Yes	Yes	Yes	N/A
Other Parsl executors	No	If executed on the same node with file system access	If tasks are executed on the same node or with access to the same file system	N/A

3.2 Apps

An **app** is a Parsl construct for representing a fragment of Python code or external Bash shell code that can be asynchronously executed.

A Parsl app is defined by annotating a Python function with a decorator: the `@python_app` decorator for a **Python app**, the `@bash_app` decorator for a **Bash app**, and the `@join_app` decorator for a **Join app**.

Python apps encapsulate pure Python code, while Bash apps wrap calls to external applications and scripts, and Join apps allow composition of other apps to form sub-workflows.

Python and Bash apps are documented below. Join apps are documented in a later section (see [Join Apps](#))

3.2.1 Python Apps

The following code snippet shows a Python function `double(x: int)`, which returns double the input value. The `@python_app` decorator defines the function as a Parsl Python app.

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

double(42)
```

As a Parsl Python app is executed asynchronously, and potentially remotely, the function cannot assume access to shared program state. For example, it must explicitly import any required modules and cannot refer to variables used outside the function. Thus while the following code fragment is valid Python, it is not valid Parsl, as the `bad_double()` function requires the `random` module and refers to the external variable `factor`.

```
import random
factor = 5

@python_app
def bad_double(x):
    return x * random.random() * factor

print(bad_double(42))
```

The following alternative formulation is valid Parsl.

```
import random
factor = 5

@python_app
def good_double(x, f):
    import random
    return x * random.random() * f

print(good_double(42, factor))
```

Python apps may be passed any Python input argument, including primitive types, files, and other complex types that can be serialized (e.g., numpy array, scikit-learn model). They may also be passed a Parsl `Future` (see [Futures](#)) returned by another Parsl app. In this case, Parsl will establish a dependency between the two apps and will not execute the dependent app until all dependent futures are resolved. Further detail is provided in [Futures](#).

A Python app may also act upon files. In order to make Parsl aware of these files, they must be specified by using the `inputs` and/or `outputs` keyword arguments, as in following code snippet, which copies the contents of one file (`in.txt`) to another (`out.txt`).

```
@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])
```

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](#) or files. Parsl will wait for the results of any listed [Futures](#) to be resolved before executing the app. The `inputs` argument is useful both for passing files as arguments and when one wishes to pass in an arbitrary number of futures at call time.
2. `outputs`: (list) This keyword argument defines a list of files that will be produced by the app. For each file thus listed, Parsl will create a future, track the file, and ensure that it is correctly created. The future can then be passed to other apps as an input argument.
3. `walltime`: (int) This keyword argument places a limit on the app's runtime in seconds. If the walltime is exceed, Parsl will raise an `parsl.app.errors.AppTimeout` exception.

Returns

A Python app returns an `AppFuture` (see [Futures](#)) as a proxy for the results that will be returned by the app once it is executed. This future can be inspected to obtain task status; and it can be used to wait for the result, and when complete, present the output Python object(s) returned by the app. In case of an error or app failure, the future holds the exception raised by the app.

Limitations

There are some limitations on the Python functions that can be converted to apps:

1. Functions should act only on defined input arguments. That is, they should not use script-level or global variables.
2. Functions must explicitly import any required modules.
3. Parsl uses `cloudpickle` and `pickle` to serialize Python objects to/from apps. Therefore, Parsl requires that all input and output objects can be serialized by `cloudpickle` or `pickle`. See [Addressing `SerializationError`](#).
4. STDOUT and STDERR produced by Python apps remotely are not captured.

3.2.2 Bash Apps

A Parsl Bash app is used to execute an external application, script, or code written in another language. It is defined by a `@bash_app` decorator and the Python code that forms the body of the function must return a fragment of Bash shell code to be executed by Parsl. The Bash shell code executed by a Bash app can be arbitrarily long.

The following code snippet presents an example of a Bash app `echo_hello`, which returns the bash command `'echo "Hello World!"'` as a string. This string will be executed by Parsl as a Bash command.

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

# echo_hello() when called will execute the shell command and
# create a std.out file with the contents "Hello World!"
echo_hello()
```

Unlike a Python app, a Bash app cannot return Python objects. Instead, Bash apps communicate with other apps via files. A decorated `@bash_app` exposes the `inputs` and `outputs` keyword arguments described above for tracking input and output files. It also includes, as described below, keyword arguments for capturing the STDOUT and STDERR streams and recording them in files that are managed by Parsl.

Special Keywords

In addition to the `inputs`, `outputs`, and `walltime` keyword arguments described above, a Bash app can accept the following keywords:

1. `stdout`: (string, tuple 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. If set to a tuple (`filename`, `mode`), standard output will be redirected to the named file, opened with the specified mode as used by the Python `open` function.
2. `stderr`: (string or `parsl.AUTO_LOGNAME`) Like `stdout`, but for the standard error stream.
3. `label`: (string) If the app is invoked with `stdout=parsl.AUTO_LOGNAME` or `stderr=parsl.AUTO_LOGNAME`, this argument will be appended to the log name.

A Bash app can construct the Bash command string to be executed from arguments passed to the decorated function.

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

(continues on next page)

(continued from previous page)

```
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, like a Python app, returns an `AppFuture`, which can be used to obtain task status, determine when the app has completed (e.g., via `future.result()` as in the preceding code fragment), and access exceptions. As a Bash app can only return results via files specified via `outputs`, `stderr`, or `stdout`; the value returned by the `AppFuture` has no meaning.

If the Bash app exits with Unix exit code 0, then the `AppFuture` will complete. If the Bash app exits with any other code, Parsl will treat this as a failure, and the `AppFuture` will instead contain an `BashExitFailure` exception. The Unix exit code can be accessed through the `exitcode` attribute of that `BashExitFailure`.

Limitations

The following limitation applies to Bash apps:

1. Environment variables are not supported.

3.3 Futures

When an ordinary Python function is invoked in a Python program, the Python interpreter waits for the function to complete execution before proceeding to the next statement. But if a function is expected to execute for a long period of time, it may be preferable not to wait for its completion but instead to proceed immediately with executing subsequent statements. The function can then execute concurrently with that other computation.

Concurrency can be used to enhance performance when independent activities can execute on different cores or nodes in parallel. The following code fragment demonstrates this idea, showing that overall execution time may be reduced if the two function calls are executed concurrently.

```
v1 = expensive_function(1)
v2 = expensive_function(2)
result = v1 + v2
```

However, concurrency also introduces a need for **synchronization**. In the example, it is not possible to compute the sum of `v1` and `v2` until both function calls have completed. Synchronization provides a way of blocking execution of one activity (here, the statement `result = v1 + v2`) until other activities (here, the two calls to `expensive_function()`) have completed.

Parsl supports concurrency and synchronization as follows. Whenever a Parsl program calls a Parsl app (a function annotated with a Parsl app decorator, see [Apps](#)), Parsl will create a new `task` and immediately return a `future` in lieu of that function's result(s). The program will then continue immediately to the next statement in the program. At some point, for example when the task's dependencies are met and there is available computing capacity, Parsl will execute the task. Upon completion, Parsl will set the value of the future to contain the task's output.

A future can be used to track the status of an asynchronous task. For example, after creation, the future may be interrogated to determine the task's status (e.g., running, failed, completed), access results, and capture exceptions. Further, futures may be used for synchronization, enabling the calling Python program to block until the future has completed execution.

Parsl provides two types of futures: *AppFuture* and *DataFuture*. While related, they enable subtly different parallel patterns.

3.3.1 AppFutures

AppFutures are the basic building block upon which Parsl programs are built. Every invocation of a Parsl app returns an AppFuture that may be used to monitor and manage the task's execution. AppFutures are inherited from Python's *concurrent library*. They provide three key capabilities:

1. An AppFuture's `result()` function can be used to wait for an app to complete, and then access any result(s). This function is blocking: it returns only when the app completes or fails. The following code fragment implements an example similar to the `expensive_function()` example above. Here, the `sleep_double` app simply doubles the input value. The program invokes the `sleep_double` app twice, and returns futures in place of results. The example shows how the future's `result()` function can be used to wait for the results from the two `sleep_double` app invocations to be computed.

```
@python_app
def sleep_double(x):
    import time
    time.sleep(2)    # Sleep for 2 seconds
    return x*2

# Start two concurrent sleep_double apps. doubled_x1 and doubled_x2 are AppFutures
doubled_x1 = sleep_double(10)
doubled_x2 = sleep_double(5)

# The result() function will block until each of the corresponding app calls have_
↪completed
print(doubled_x1.result() + doubled_x2.result())
```

2. An AppFuture's `done()` function can be used to check the status of an app, *without blocking*. The following example shows that calling the future's `done()` function will not stop execution of the main Python program.

```
@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())
```

3. An AppFuture provides a safe way to handle exceptions and errors while asynchronously executing apps. The example shows how exceptions can be captured in the same way as a standard Python program when calling the future's `result()` function.

```
@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.
```

(continues on next page)

(continued from previous page)

```
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 upon 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 an `AppFuture` represents the execution of an asynchronous app, a `DataFuture` represents a file to be produced by that app. Parsl's dataflow model requires such a construct so that it can determine when dependent apps, apps that that are to consume a file produced by another app, can start execution.

When calling an app that produces files as outputs, Parsl requires that a list of output files be specified (as a list of `File` objects passed in via the `outputs` keyword argument). Parsl will return a `DataFuture` for each output file as part `AppFuture` when the app is executed. These `DataFutures` are accessible in the `AppFuture`'s `outputs` attribute.

Each `DataFuture` will complete when the App has finished executing, and the corresponding file has been created (and if specified, staged out).

When a `DataFuture` is passed as an argument to a subsequent app invocation, that subsequent app will not begin execution until the `DataFuture` is completed. The input argument will then be replaced with an appropriate `File` object.

The following code snippet shows how `DataFutures` are used. In this example, the call to the `echo Bash` app specifies that the results should be written to an output file ("hello1.txt"). The main program inspects the status of the output file (via the future's `outputs` attribute) and then blocks waiting for the file to be created (`hello.outputs[0].result()`).

```
# 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=[File('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 be passed to `open` directly.

3.4 Passing Python objects

Parsl apps can communicate via standard Python function parameter passing and return statements. The following example shows how a Python string can be passed to, and returned from, a Parsl app.

```
@python_app
def example(name):
    return 'hello {0}'.format(name)

r = example('bob')
print(r.result())
```

Parsl uses the cloudpickle and pickle libraries to serialize Python objects into a sequence of bytes that can be passed over a network from the submitting machine to executing workers.

Thus, Parsl apps can receive and return standard Python data types such as booleans, integers, tuples, lists, and dictionaries. However, not all objects can be serialized with these methods (e.g., closures, generators, and system objects), and so those objects cannot be used with all executors.

Parsl will raise a *SerializationError* if it encounters an object that it cannot serialize. This applies to objects passed as arguments to an app, as well as objects returned from an app. See *Addressing SerializationError*.

3.5 Staging data files

Parsl apps can take and return data files. A file may be passed as an input argument to an app, or returned from an app after execution. Parsl provides support to automatically transfer (stage) files between the main Parsl program, worker nodes, and external data storage systems.

Input files can be passed as regular arguments, or a list of them may be specified in the special `inputs` keyword argument to an app invocation.

Inside an app, the `filepath` attribute of a *File* can be read to determine where on the execution-side file system the input file has been placed.

Output *File* objects must also be passed in at app invocation, through the `outputs` parameter. In this case, the *File* object specifies where Parsl should place output after execution.

Inside an app, the `filepath` attribute of an output *File* provides the path at which the corresponding output file should be placed so that Parsl can find it after execution.

If the output from an app is to be used as the input to a subsequent app, then a *DataFuture* that represents whether the output file has been created must be extracted from the first app's *AppFuture*, and that must be passed to the second app. This causes app executions to be properly ordered, in the same way that passing *AppFutures* to subsequent apps causes execution ordering based on an app returning.

In a Parsl program, file handling is split into two pieces: files are named in an execution-location independent manner using *File* objects, and executors are configured to stage those files in to and out of execution locations using instances of the *Staging* interface.

3.5.1 Parsl files

Parsl uses a custom *File* to provide a location-independent way of referencing and accessing files. Parsl files are defined by specifying the URL *scheme* and a path to the file. Thus a file may represent an absolute path on the submit-side file system or a URL to an external file.

The scheme defines the protocol via which the file may be accessed. Parsl supports the following schemes: file, ftp, http, https, and globus. If no scheme is specified Parsl will default to the file scheme.

The following example shows creation of two files with different schemes: a locally-accessible data.txt file and an HTTPS-accessible README file.

```
File('file://home/parsl/data.txt')
File('https://github.com/Parsl/parsl/blob/master/README.rst')
```

Parsl automatically translates the file's location relative to the environment in which it is accessed (e.g., the Parsl program or an app). The following example shows how a file can be accessed in the app irrespective of where that app executes.

```
@python_app
def print_file(inputs=[]):
    with open(inputs[0].filepath, 'r') as inp:
        content = inp.read()
        return(content)

# create an remote Parsl file
f = File('https://github.com/Parsl/parsl/blob/master/README.rst')

# call the print_file app with the Parsl file
r = print_file(inputs=[f])
r.result()
```

As described below, the method by which this files are transferred depends on the scheme and the staging providers specified in the Parsl configuration.

3.5.2 Staging providers

Parsl is able to transparently stage files between at-rest locations and execution locations by specifying a list of *Staging* instances for an executor. These staging instances define how to transfer files in and out of an execution location. This list should be supplied as the `storage_access` parameter to an executor when it is constructed.

Parsl includes several staging providers for moving files using the schemes defined above. By default, Parsl executors are created with three common staging providers: the NoOpFileStaging provider for local and shared file systems and the HTTP(S) and FTP staging providers for transferring files to and from remote storage locations. The following example shows how to explicitly set the default staging providers.

```
from parsl.config import Config
from parsl.executors import HighThroughputExecutor
from parsl.data_provider.data_manager import default_staging

config = Config(
    executors=[
        HighThroughputExecutor(
            storage_access=default_staging,
            # equivalent to the following
            # storage_access=[NoOpFileStaging(), FTPSeparateTaskStaging(),
            ↪ HTTPSeparateTaskStaging()],
```

(continues on next page)

(continued from previous page)

```

    )
]
)

```

Parsl further differentiates when staging occurs relative to the app invocation that requires or produces files. Staging either occurs with the executing task (*in-task staging*) or as a separate task (*separate task staging*) before app execution. In-task staging uses a wrapper that is executed around the Parsl task and thus occurs on the resource on which the task is executed. Separate task staging inserts a new Parsl task in the graph and associates a dependency between the staging task and the task that depends on that file. Separate task staging may occur on either the submit-side (e.g., when using Globus) or on the execution-side (e.g., HTTPS, FTP).

NoOpFileStaging for Local/Shared File Systems

The NoOpFileStaging provider assumes that files specified either with a path or with the `file` URL scheme are available both on the submit and execution side. This occurs, for example, when there is a shared file system. In this case, files will not be moved, and the File object simply presents the same file path to the Parsl program and any executing tasks.

Files defined as follows will be handled by the NoOpFileStaging provider.

```

File('file://home/parsl/data.txt')
File('/home/parsl/data.txt')

```

The NoOpFileStaging provider is enabled by default on all executors. It can be explicitly set as the only staging provider as follows.

```

from parsl.config import Config
from parsl.executors import HighThroughputExecutor
from parsl.data_provider.file_noop import NoOpFileStaging

config = Config(
    executors=[
        HighThroughputExecutor(
            storage_access=[NoOpFileStaging()]
        )
    ]
)

```

FTP, HTTP, HTTPS: separate task staging

Files named with the `ftp`, `http` or `https` URL scheme will be staged in using HTTP GET or anonymous FTP commands. These commands will be executed as a separate Parsl task that will complete before the corresponding app executes. These providers cannot be used to stage out output files.

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 whatever location is selected for the app to execute. The following example illustrates how the remote file is implicitly downloaded from an FTP server and then converted. Note that the app does not need to know the location of the downloaded file on the remote computer, 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')

# 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()
```

HTTP and FTP separate task staging providers can be configured as follows.

```
from parsl.config import Config
from parsl.executors import HighThroughputExecutor
from parsl.data_provider.http import HTTPSeparateTaskStaging
from parsl.data_provider.ftp import FTPSeparateTaskStaging

config = Config(
    executors=[
        HighThroughputExecutor(
            storage_access=[HTTPSeparateTaskStaging(), FTPSeparateTaskStaging()]
        )
    ]
)
```

FTP, HTTP, HTTPS: in-task staging

These staging providers are intended for use on executors that do not have a file system shared between each executor node.

These providers will use the same HTTP GET/anonymous FTP as the separate task staging providers described above, but will do so in a wrapper around individual app invocations, which guarantees that they will stage files to a file system visible to the app.

A downside of this staging approach is that the staging tasks are less visible to Parsl, as they are not performed as separate Parsl tasks.

In-task staging providers can be configured as follows.

```
from parsl.config import Config
from parsl.executors import HighThroughputExecutor
from parsl.data_provider.http import HTTPInTaskStaging
from parsl.data_provider.ftp import FTPInTaskStaging

config = Config(
    executors=[
        HighThroughputExecutor(
            storage_access=[HTTPInTaskStaging(), FTPInTaskStaging()]
        )
    ]
)
```

(continues on next page)

(continued from previous page)

```
]
)
```

Globus

The Globus staging provider is used to transfer 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: a Globus endpoint's UUID can be found in the Globus [Manage Endpoints](#) page.

There must also be a Globus endpoint available with access to a execute-side file system, because Globus file transfers happen between two Globus endpoints.

Globus Configuration

In order to manage where files are staged, users must configure the default `working_dir` on a remote location. This information is specified in the `ParslExecutor` via the `working_dir` parameter in the `Config` instance. For example:

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

config = Config(
    executors=[
        HighThroughputExecutor(
            working_dir="/home/user/data"
        )
    ]
)
```

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), as in the following. In most cases, `endpoint_path` and `local_path` are the same and do not need to be specified.

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

config = Config(
    executors=[
        HighThroughputExecutor(
            working_dir="/home/user/parsl_script",
            storage_access=default_staging + [GlobusStaging(
```

(continues on next page)

(continued from previous page)

```

        endpoint_uuid="7d2dc622-2edb-11e8-b8be-0ac6873fc732",
        endpoint_path="/",
        local_path="/home/user"
    )]
)
)

```

Globus Authorization

In order to transfer files with Globus, the user must first authenticate. The first time that Globus is used with Parsl on a computer, the program will prompt the user to follow an authentication and authorization procedure involving a web browser. Users can authorize out of band by running the `parsl-globus-auth` utility. This is useful, for example, when running a Parsl program in a batch system where it will be unattended.

```

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

```

rsync

The `rsync` utility can be used to transfer files in the `file` scheme in configurations where workers cannot access the submit-side file system directly, such as when executing on an AWS EC2 instance or on a cluster without a shared file system. However, the submit-side file system must be exposed using `rsync`.

rsync Configuration

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

An `RSyncStaging` option must then be added to the Parsl configuration file, as in the following. 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)],
            ...
        )
    )
)

```

rsync 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 submit-side system. The form of this authorization will depend on the systems in question.

The following example installs an ssh key from the submit-side file system 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 submit-side 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
                ...
            )
        )
    ]
)
```

3.6 Execution

Contemporary computing environments may include a wide range of computational platforms or **execution providers**, from laptops and PCs to various clusters, supercomputers, and cloud computing platforms. Different execution providers may require or allow for the use of different **execution models**, such as threads (for efficient parallel execution on a multicore processor), processes, and pilot jobs for running many small tasks on a large parallel system.

Parsl is designed to abstract these low-level details so that an identical Parsl program can run unchanged on different platforms or across multiple platforms. To this end, Parsl uses a configuration file to specify which execution provider(s) and execution model(s) to use. Parsl provides a high level abstraction, called a *block*, for providing a uniform description of a compute resource irrespective of the specific execution provider.

Note: Refer to [Configuration](#) for information on how to configure the various components described below for specific scenarios.

3.6.1 Execution providers

Clouds, supercomputers, and local PCs offer vastly different modes of access. To overcome these differences, and present a single uniform interface, Parsl implements a simple provider abstraction. This abstraction is key to Parsl's ability to enable scripts to be moved between resources. The provider interface exposes three core actions: submit a job for execution (e.g., `sbatch` for the Slurm resource manager), retrieve the status of an allocation (e.g., `squeue`), and cancel a running job (e.g., `scancel`). Parsl implements providers for local execution (`fork`), for various cloud platforms using cloud-specific APIs, and for clusters and supercomputers that use a Local Resource Manager (LRM) to manage access to resources, such as Slurm, HTCondor, and Cobalt.

Each provider implementation may allow users to specify additional parameters for further configuration. Parameters are generally mapped to LRM submission script or cloud API options. Examples of LRM-specific options are partition, wall clock time, scheduler options (e.g., `#SBATCH` arguments for Slurm), and worker initialization commands (e.g., loading a conda environment). Cloud parameters include access keys, instance type, and spot bid price

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. *AWSPProvider*: 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. *KubernetesProvider*: This provider allows you to provision and manage containers on a Kubernetes cluster.
10. *AdHocProvider*: This provider allows you manage execution over a collection of nodes to form an ad-hoc cluster.
11. *LSFProvider*: This provider allows you to schedule resources via IBM's LSF scheduler

3.6.2 Executors

Parsl programs vary widely in terms of their execution requirements. Individual Apps may run for milliseconds or days, and available parallelism can vary between none for sequential programs to millions for “pleasingly parallel” programs. Parsl executors, as the name suggests, execute Apps on one or more target execution resources such as multi-core workstations, clouds, or supercomputers. As it appears infeasible to implement a single execution strategy that will meet so many diverse requirements on such varied platforms, Parsl provides a modular executor interface and a collection of executors that are tuned for common execution patterns.

Parsl executors extend the Executor class offered by Python's `concurrent.futures` library, which allows Parsl to use existing solutions in the Python Standard Library (e.g., `ThreadPoolExecutor`) and from other packages such as `IPyParallel`. Parsl extends the `concurrent.futures` executor interface to support additional capabilities such as automatic

scaling of execution resources, monitoring, deferred initialization, and methods to set working directories. All executors share a common execution kernel that is responsible for deserializing the task (i.e., the App and its input arguments) and executing the task in a sandboxed Python environment.

Parsl currently supports the following executors:

1. *ThreadPoolExecutor*: This executor supports multi-thread execution on local resources.
2. *HighThroughputExecutor*: This executor implements hierarchical scheduling and batching using a pilot job model to deliver high throughput task execution on up to 4000 Nodes.
3. *WorkQueueExecutor*: **[Beta]** This executor 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. *ExtremeScaleExecutor*: **[Beta]** The ExtremeScaleExecutor uses [mpi4py](#) to scale to 4000+ nodes. This executor is typically used for executing 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

Many LRMs offer mechanisms for spawning applications across nodes inside a single job and for specifying the resources and task placement information needed to execute that application at launch time. Common mechanisms include [srun](#) (for Slurm), [aprun](#) (for Crays), and [mpirun](#) (for MPI). Thus, to run Parsl programs on such systems, we typically want first to request a large number of nodes and then to *launch* “pilot job” or **worker** processes using the system launchers. Parsl’s Launcher abstraction enables Parsl programs to use these system-specific launcher systems to start 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, the launcher interface can be used to implement specialized behaviors in custom environments (for example, to launch node processes inside containers with customized environments). For example, the following 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()

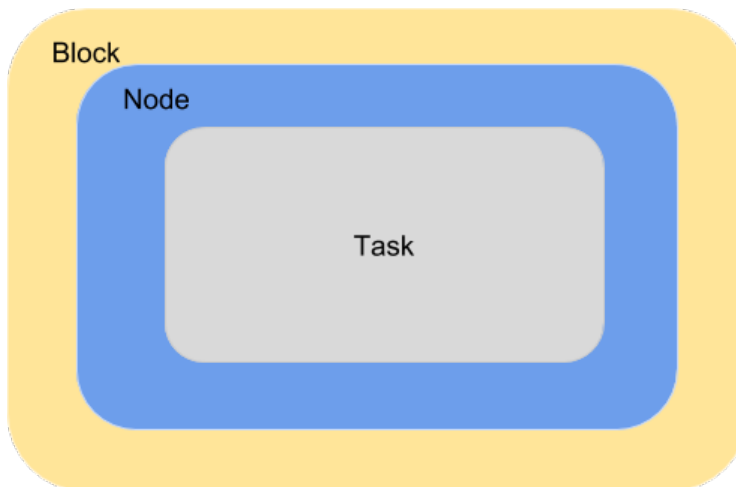
    def __call__(self, command, tasks_per_node, nodes_per_block):
        new_command="worker-wrapper {}".format(command)
        return self.srun_launcher(new_command, tasks_per_node, nodes_per_block)
```

3.6.4 Blocks

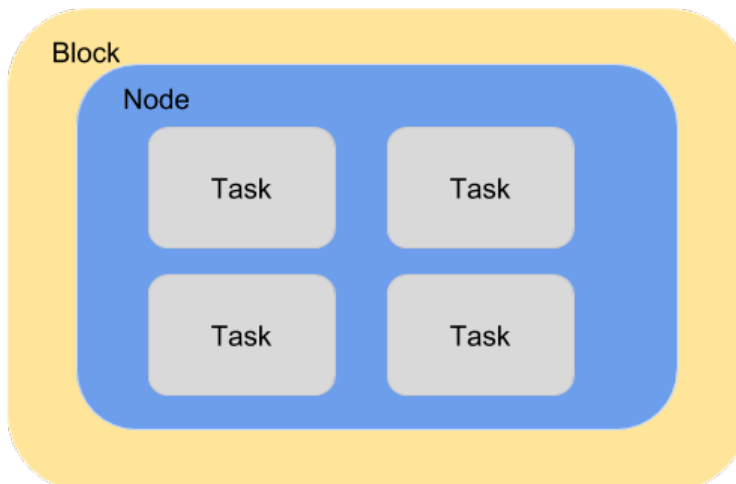
One challenge when making use of heterogeneous execution resource types is the need to provide a uniform representation of resources. Consider that single requests on clouds return individual nodes, clusters and supercomputers provide batches of nodes, grids provide cores, and workstations provide a single multicore node

Parsl defines a resource abstraction called a *block* as the most basic unit of resources to be acquired from a provider. A block contains one or more nodes and maps to the different provider abstractions. In a cluster, a block corresponds to a single allocation request to a scheduler. In a cloud, a block corresponds to a single API request for one or more instances. Parsl can then execute *tasks* (instances of apps) within and across (e.g., for MPI jobs) nodes within a block. Blocks are also used as the basis for elasticity on batch scheduling systems (see Elasticity below). Three different examples of block configurations are shown below.

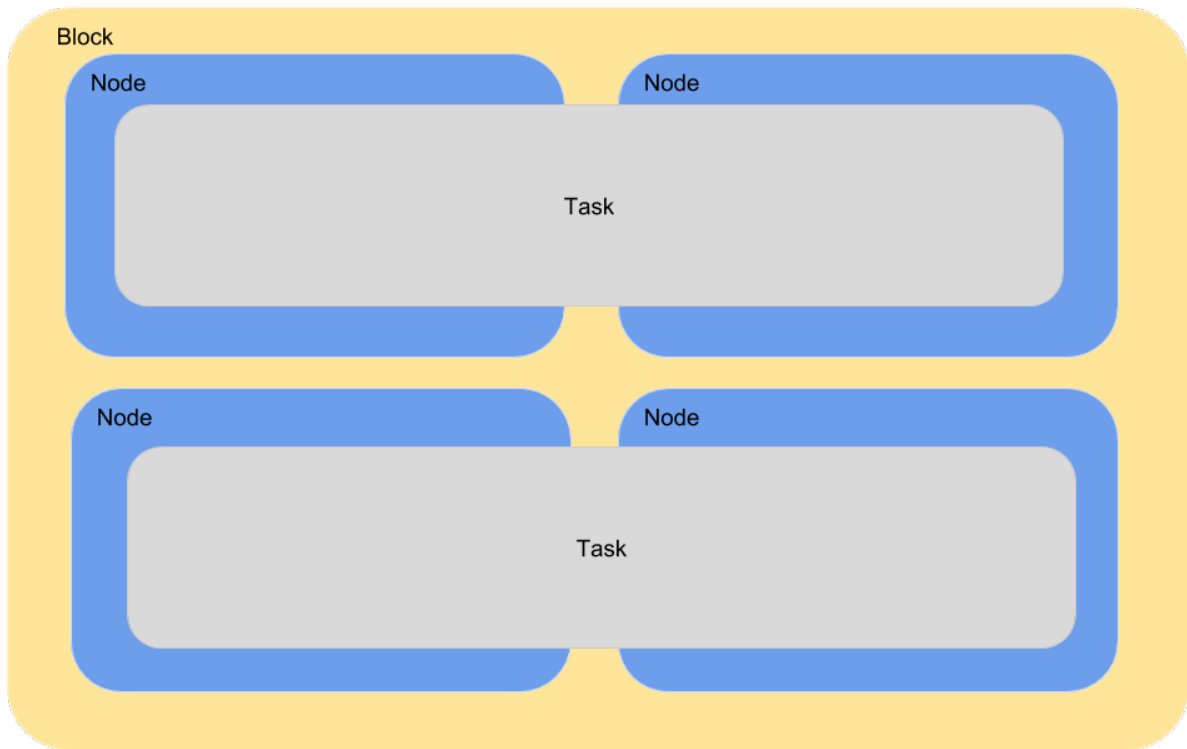
1. A single block comprised of a node executing one task:



2. A single block with one 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).



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.

3.6.5 Elasticity

Workload resource requirements often vary over time. For example, in the map-reduce paradigm the map phase may require more resources than the reduce phase. In general, reserving sufficient resources for the widest parallelism will result in underutilization during periods of lower load; conversely, reserving minimal resources for the thinnest parallelism will lead to optimal utilization but also extended execution time. Even simple bag-of-task applications may have tasks of different durations, leading to trailing tasks with a thin workload.

To address dynamic workload requirements, Parsl implements a cloud-like elasticity model in which resource blocks are provisioned/deprovisioned in response to workload pressure. Parsl provides an extensible strategy interface by which users can implement their own elasticity logic. Given the general nature of the implementation, Parsl can provide elastic execution on clouds, clusters, and supercomputers. Of course, in an HPC setting, elasticity may be complicated by queue delays.

Parsl's elasticity model 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 workload needs.

The animated diagram below shows how blocks are elastically managed within an executor. The Parsl 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.

Parallelism

Parsl provides a user-managed model for controlling elasticity. In addition to setting the minimum and maximum number of blocks to be provisioned, users can also define the desired level of parallelism by setting a parameter (p). Parallelism is expressed as the ratio of task execution capacity to 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 the maximum resources needed are used (i.e., `max_blocks`); 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 provisioning aggressiveness can be controlled.

For example:

- When $p = 0$: Use the fewest resources possible. If there is no workload then no blocks will be provisioned, otherwise the fewest blocks specified (e.g., `min_blocks`, or 1 if `min_blocks` is set to 0) will be provisioned.

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

- When $p = 1$: Use as many resources as possible. Provision sufficient nodes to execute all running and available tasks concurrently up to the `max_blocks` specified.

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

- When $p = 1/2$: Queue up to 2 tasks per worker before requesting a new block.

Configuration

The example below shows how elasticity and parallelism can be configured. Here, a `HighThroughputExecutor` is used with a minimum of 1 block and a maximum of 2 blocks, where each block may host up to 2 workers per node. Thus this setup is capable of servicing 2 tasks concurrently. Parallelism of 0.5 means that when more than $2 * \text{total task capacity}$ (i.e., 4 tasks) are queued a new block will be requested. An example `Config` is:

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

config = Config(
    executors=[
        HighThroughputExecutor(
            label='local_htex',
            workers_per_node=2,
            provider=Local(
                min_blocks=1,
                init_blocks=1,
                max_blocks=2,
                nodes_per_block=1,
                parallelism=0.5
            )
        )
    ]
)
```

(continues on next page)

(continued from previous page)

```

    )
  ]
)

```

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 use of one or more executors as specified in the configuration. In this situation, individual apps may indicate which executors they are able to use.

The common scenarios for this feature are:

- A 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.
- A 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 a login node.
- A 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 other apps. Often, 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 apps can specify suitable executors 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

Parsl provides various mechanisms to add resiliency and robustness to programs.

3.7.1 Exceptions

Parsl is designed to capture, track, and handle various errors occurring during execution, including those related to the program, apps, execution environment, and Parsl itself. It also provides functionality to appropriately respond to failures during execution.

Failures might occur for various reasons:

1. A task failed during execution.
2. A task failed to launch, for example, because an input dependency was not met.
3. There was a formatting error while formatting the command-line string in Bash apps.
4. A task completed execution but failed to produce one or more of its specified outputs.
5. Task exceeded the specified walltime.

Since Parsl tasks are executed asynchronously and remotely, it can be difficult to determine when errors have occurred and to appropriately handle them in a Parsl program.

For errors occurring in Python code, Parsl captures Python exceptions and returns them to the main Parsl program. For non-Python errors, for example when a node or worker fails, Parsl imposes a timeout, and considers a task to have failed if it has not heard from the task by that timeout. Parsl also considers a task to have failed if it does not meet the contract stated by the user during invocation, such as failing to produce the stated output files.

Parsl communicates these errors by associating Python exceptions with 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. In these cases, retrying failed tasks can be a simple way of overcoming transient (e.g., machine failure, network failure) and intermittent failures. When `retries` are enabled (and set to an integer > 0), Parsl will automatically re-launch tasks that have failed until the retry limit is reached. By default, retries are disabled and exceptions will be communicated to the Parsl program.

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

```
import parsl
from parsl.configs.htex_local import config

config.retries = 2

parsl.load(config)
```

3.7.3 Lazy fail

Parsl implements a lazy failure model through which a workload will continue to execute in the case that some tasks fail. That is, the program will not halt as soon as it encounters a failure, rather it will continue to execute unaffected apps.

The following example shows how lazy failures affect execution. In this case, task C fails and therefore tasks E and F that depend on results from C cannot be executed; however, Parsl will continue to execute tasks B and D as they are unaffected by task C's failure.

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 Memoization and checkpointing

When an app is invoked several times with the same parameters, Parsl can reuse the result from the first invocation without executing the app again.

This can save time and computational resources.

This is done in two ways:

- Firstly, *app caching* will allow reuse of results within the same run.
- Building on top of that, *checkpointing* will store results on the filesystem and reuse those results in later runs.

3.8.1 App caching

There are many situations in which a program may be re-executed over time. Often, large fragments of the program will not have changed and therefore, re-execution of apps will waste valuable time and computation resources. Parsl's app caching solves this problem by storing results from apps that have successfully completed so that they can be re-used.

App caching is enabled by setting the `cache` argument in the `python_app()` or `bash_app()` decorator to `True` (by default it is `False`).

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

App caching can be globally disabled by setting `app_cache=False` in the `Config`.

App caching can be particularly useful when developing interactive programs 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.

App equivalence

Parsl determines app equivalence by storing the a hash of the app function. Thus, any changes to the app code (e.g., its signature, its body, or even the docstring within the body) will invalidate cached values.

However, Parsl does not traverse the call graph of the app function, so changes inside functions called by an app will not invalidate cached values.

Invocation equivalence

Two app invocations are determined to be equivalent if their input arguments are identical.

In simple cases, this follows obvious rules:

```
# these two app invocations are the same and the second invocation will
# reuse any cached input from the first invocation
x = 7
f(x).result()

y = 7
f(y).result()
```

Internally, equivalence is determined by hashing the input arguments, and comparing the hash to hashes from previous app executions.

This approach can only be applied to data types for which a deterministic hash can be computed.

By default Parsl can compute sensible hashes for basic data types: `str`, `int`, `float`, `None`, as well as more some complex types: functions, and dictionaries and lists containing hashable types.

Attempting to cache apps invoked with other, non-hashable, data types will lead to an exception at invocation.

In that case, mechanisms to hash new types can be registered by a program by implementing the `parsl.dataflow.memoization.id_for_memo` function for the new type.

Ignoring arguments

On occasion one may wish to ignore particular arguments when determining app invocation equivalence - for example, when generating log file names automatically based on time or run information. Parsl allows developers to list the arguments to be ignored in the `ignore_for_cache` app decorator parameter:

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

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 obscure 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 an app are made concurrently having not yet cached a result, many instances of the app will be launched. Once one invocation completes and the result is cached all subsequent calls will return immediately with the cached result.
- **Performance:** If app caching is enabled, there may be some performance overhead especially if a large number of short duration tasks are launched rapidly. This overhead has not been quantified.

3.8.2 Checkpointing

Large-scale Parsl programs are likely to encounter errors due to node failures, application or environment errors, and myriad other issues. Parsl offers an application-level checkpointing model to improve resilience, fault tolerance, and efficiency.

Note: Checkpointing builds on top of app caching, and so app caching must be enabled. If app caching 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 a Parsl program loads a checkpoint file and is executed, it will use checkpointed results for any apps that have been previously executed. Like app caching, checkpoints use the hash of the app and the invocation input parameters to identify 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. Results will be saved to the checkpoint file for all tasks that have completed during this period.

```
>>> 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 program termination from exceptions, terminate signals, etc. However it is still possible that information might be lost if the program 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 Parsl program 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.

Note: Checkpoint modes `periodic`, `dfk_exit`, and `manual` can interfere with garbage collection. In these modes task information will be retained after completion, until checkpointing events are triggered.

Creating a checkpoint

Automated checkpointing must be explicitly enabled in the Parsl configuration. There is no need to modify a Parsl program as checkpointing will occur transparently. In the following example, checkpointing is enabled at task exit. The results of each invocation of the `slow_double` app will be stored in the checkpoint file.

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

config.checkpoint_mode = 'task_exit'

parsl.load(config)

@python_app(cache=True)
def slow_double(x):
    import time
    time.sleep(5)
    return x * 2

d = []
for i in range(5):
    d.append(slow_double(i))

print([d[i].result() for i in range(5)])
```

Alternatively, manual checkpointing can be used to explicitly specify when the checkpoint file should be saved. The following example shows how manual checkpointing can be used. Here, the `dfk.checkpoint()` function will save the results of the prior invocations of the `slow_double` app.

```

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

```

Resuming from a checkpoint

When resuming a program from a checkpoint Parsl allows the user to select which checkpoint file(s) to use. Checkpoint files are stored in the `runinfo/RUNID/checkpoint` directory.

The example below shows how to resume using all available checkpoints. Here, the program re-executes the same calls to the `slow_double` app as above and instead of waiting for results to be computed, the values from the checkpoint file are immediately returned.

```

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)

# Rerun the same workflow
d = []
for i in range(5):
    d.append(slow_double(i))

# wait for results
print([d[i].result() for i in range(5)])

```

3.9 Configuration

Parsl separates program logic from execution configuration, enabling programs to be developed entirely independently from their execution environment. Configuration is described by a Python object (*Config*) so that developers can introspect permissible options, validate settings, and retrieve/edit configurations dynamically during execution. A configuration object specifies details of the provider, executors, connection channel, allocation size, queues, durations, and data management options.

The following example shows a basic configuration object (*Config*) for the Frontera supercomputer at TACC. This config uses the *HighThroughputExecutor* to submit tasks from a login node (*LocalChannel*). It requests an allocation of 128 nodes, deploying 1 worker for each of the 56 cores per node, from the normal partition. The config uses the *address_by_hostname()* helper function to determine the login node's IP address.

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

config = Config(
    executors=[
        HighThroughputExecutor(
            label="frontera_htex",
            address=address_by_hostname(),
            max_workers=56,
            provider=SlurmProvider(
                channel=LocalChannel(),
                nodes_per_block=128,
                init_blocks=1,
                partition='normal',
                launcher=SrunLauncher(),
            ),
        ),
    ],
)
```

Configuration How-To and Examples:

- *Configuration*
 - *How to Configure*
 - *Heterogeneous Resources*
 - *Ad-Hoc Clusters*
 - *Amazon Web Services*
 - *ASPIRE 1 (NSCC)*
 - *Blue Waters (NCSA)*
 - *Bridges (PSC)*
 - *CC-IN2P3*
 - *CCL (Notre Dame, with Work Queue)*

- *Comet (SDSC)*
- *Cooley (ALCF)*
- *Cori (NERSC)*
- *Frontera (TACC)*
- *Kubernetes Clusters*
- *Midway (RCC, UChicago)*
- *Open Science Grid*
- *Stampede2 (TACC)*
- *Summit (ORNL)*
- *Theta (ALCF)*
- *Further help*

Note: All configuration examples below must be customized for the user's allocation, Python environment, file system, etc.

3.9.1 How to Configure

The configuration specifies what, and how, resources are to be used for executing the Parsl program and its apps. It is important to carefully consider the needs of the Parsl program and its apps, and the characteristics of the compute resources, to determine an ideal configuration. Aspects to consider include: 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 Parsl request multiple nodes in an individual scheduler job; and 4) where will the main Parsl program run and how will it communicate with the apps.

Stepping through the following question should help formulate a suitable configuration object.

1. Where should apps be executed?

Target	Executor	Provider
Laptop/Workstation	<ul style="list-style-type: none"> • <i>HighThroughputExecutor</i> • <i>ThreadPoolExecutor</i> • <i>WorkQueueExecutor</i> <i>beta</i> 	<i>LocalProvider</i>
Amazon Web Services	<ul style="list-style-type: none"> • <i>HighThroughputExecutor</i> 	<i>AWSProvider</i>
Google Cloud	<ul style="list-style-type: none"> • <i>HighThroughputExecutor</i> 	<i>GoogleCloudProvider</i>
Slurm based system	<ul style="list-style-type: none"> • <i>ExtremeScaleExecutor</i> • <i>HighThroughputExecutor</i> • <i>WorkQueueExecutor</i> <i>beta</i> 	<i>SlurmProvider</i>
Torque/PBS based system	<ul style="list-style-type: none"> • <i>ExtremeScaleExecutor</i> • <i>HighThroughputExecutor</i> • <i>WorkQueueExecutor</i> <i>beta</i> 	<i>TorqueProvider</i>
Cobalt based system	<ul style="list-style-type: none"> • <i>ExtremeScaleExecutor</i> • <i>HighThroughputExecutor</i> • <i>WorkQueueExecutor</i> <i>beta</i> 	<i>CobaltProvider</i>
GridEngine based system	<ul style="list-style-type: none"> • <i>HighThroughputExecutor</i> • <i>WorkQueueExecutor</i> <i>beta</i> 	<i>GridEngineProvider</i>
Condor based cluster or grid	<ul style="list-style-type: none"> • <i>HighThroughputExecutor</i> • <i>WorkQueueExecutor</i> <i>beta</i> 	<i>CondorProvider</i>
Kubernetes cluster	<ul style="list-style-type: none"> • <i>HighThroughputExecutor</i> 	<i>KubernetesProvider</i>

WorkQueueExecutor is available in v1.0.0 in beta status.

2. How many nodes will be used to execute the apps? What task durations are necessary to achieve good performance?

Executor	Number of Nodes* ⁰	Task duration for good performance
* <i>ThreadPoolExecutor</i>	1 (Only local)	Any
<i>HighThroughputExecutor</i>	<=2000	Task duration(s)/#nodes >= 0.01 longer tasks needed at higher scale
<i>ExtremeScaleExecutor</i>	>1000, <=8000 ^{†0}	>minutes
<i>WorkQueueExecutor</i>	<=1000 ^{‡0}	10s+

Warning: *IPyParallelExecutor* is deprecated as of Parsl v0.8.0. *HighThroughputExecutor* is the recommended replacement.

3. Should Parsl request multiple nodes in an individual scheduler job? (Here the term block is equivalent to a single scheduler job.)

nodes_per_block = 1		
Provider	Executor choice	Suitable Launchers
Systems that don't use Aprun	Any	<ul style="list-style-type: none"> • <i>SingleNodeLauncher</i> • <i>SimpleLauncher</i>
Aprun based systems	Any	<ul style="list-style-type: none"> • <i>AprunLauncher</i>

nodes_per_block > 1		
Provider	Executor choice	Suitable Launchers
<i>TorqueProvider</i>	Any	<ul style="list-style-type: none"> • <i>AprunLauncher</i> • <i>MpiExecLauncher</i>
<i>CobaltProvider</i>	Any	<ul style="list-style-type: none"> • <i>AprunLauncher</i>
<i>SlurmProvider</i>	Any	<ul style="list-style-type: none"> • <i>SrunLauncher</i> if native slurm • <i>AprunLauncher</i>, otherwise

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

- 4) Where will the main Parsl program run and how will it communicate with the apps?

⁰ Assuming 32 workers per node. If there are fewer workers launched per node, a larger number of nodes could be supported.

[†] 8,000 nodes with 32 workers (256,000 workers) is the maximum scale at which the *ExtremeScaleExecutor* has been tested.

[‡] The maximum number of nodes tested for the *WorkQueueExecutor* is 10,000 GPU cores and 20,000 CPU cores.

Parsl program location	App execution target	Suitable channel
Laptop/Workstation	Laptop/Workstation	<i>LocalChannel</i>
Laptop/Workstation	Cloud Resources	No channel is needed
Laptop/Workstation	Clusters with no 2FA	<i>SSHChannel</i>
Laptop/Workstation	Clusters with 2FA	<i>SSHInteractiveLoginChannel</i>
Login node	Cluster/Supercomputer	<i>LocalChannel</i>

3.9.2 Heterogeneous Resources

In some cases, it can be difficult to specify the resource requirements for running a workflow. For example, if the compute nodes a site provides are not uniform, there is no “correct” resource configuration; the amount of parallelism depends on which node (large or small) each job runs on. In addition, the software and filesystem setup can vary from node to node. A Condor cluster may not provide shared filesystem access at all, and may include nodes with a variety of Python versions and available libraries.

The *WorkQueueExecutor* provides several features to work with heterogeneous resources. By default, Parsl only runs one app at a time on each worker node. However, it is possible to specify the requirements for a particular app, and Work Queue will automatically run as many parallel instances as possible on each node. Work Queue automatically detects the amount of cores, memory, and other resources available on each execution node. To activate this feature, add a resource specification to your apps. A resource specification is a dictionary with the following three (case-insensitive) keys: `cores` (an integer corresponding to the number of cores required by the task), `memory` (an integer corresponding to the task’s memory requirement in MB), and `disk` (an integer corresponding to the task’s disk requirement in MB), passed to an app via the special keyword argument `parsl_resource_specification`. The specification can be set for all app invocations via a default, for example:

```
@python_app
def compute(x, parsl_resource_specification={'cores': 1, 'memory': 1000,
↪ 'disk': 1000}):
    return x*2
```

or updated when the app is invoked:

```
spec = {'cores': 1, 'memory': 500, 'disk': 500}
future = compute(x, parsl_resource_specification=spec)
```

This `parsl_resource_specification` special keyword argument will inform Work Queue about the resources this app requires. When placing instances of `compute(x)`, Work Queue will run as many parallel instances as possible based on each worker node’s available resources.

If an app’s resource requirements are not known in advance, Work Queue has an auto-labeling feature that measures the actual resource usage of your apps and automatically chooses resource labels for you. With auto-labeling, it is not necessary to provide `parsl_resource_specification`; Work Queue collects stats in the background and updates resource labels as your workflow runs. To activate this feature, add the following flags to your executor config:

```
config = Config(
    executors=[
        WorkQueueExecutor(
            # ...other options go here
            autolabel=True,
            autocategory=True
        )
    ]
)
```


The `autolabel` flag tells Work Queue to automatically generate resource labels. By default, these labels are shared across all apps in your workflow. The `autocategory` flag puts each app into a different category, so that Work Queue will choose separate resource requirements for each app. This is important if e.g. some of your apps use a single core and some apps require multiple cores. Unless you know that all apps have uniform resource requirements, you should turn on `autocategory` when using `autolabel`.

The Work Queue executor can also help deal with sites that have non-uniform software environments across nodes. Parsl assumes that the Parsl program and the compute nodes all use the same Python version. In addition, any packages your apps import must be available on compute nodes. If no shared filesystem is available or if node configuration varies, this can lead to difficult-to-trace execution problems.

If your Parsl program is running in a Conda environment, the Work Queue executor can automatically scan the imports in your apps, create a self-contained software package, transfer the software package to worker nodes, and run your code inside the packaged and uniform environment. First, make sure that the Conda environment is active and you have the required packages installed (via either `pip` or `conda`):

- `python`
- `parsl`
- `ndcctools`
- `conda-pack`

Then add the following to your config:

```
config = Config(
    executors=[
        WorkQueueExecutor(
            # ...other options go here
            pack=True
        )
    ]
)
```

Note: There will be a noticeable delay the first time Work Queue sees an app; it is creating and packaging a complete Python environment. This packaged environment is cached, so subsequent app invocations should be much faster.

Using this approach, it is possible to run Parsl applications on nodes that don't have Python available at all. The packaged environment includes a Python interpreter, and Work Queue does not require Python to run.

Note: The automatic packaging feature only supports packages installed via `pip` or `conda`. Importing from other locations (e.g. via `$PYTHONPATH`) or importing other modules in the same directory is not supported.

3.9.3 Ad-Hoc Clusters

Any collection of compute nodes without a scheduler can be considered an ad-hoc cluster. Often these machines have a shared file system 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.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,
            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,
                                    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 in [issue #941](#).

3.9.4 Amazon Web Services



Note: To use AWS with Parsl, install Parsl with AWS dependencies via `python3 -m pip install parsl[aws]`

Amazon Web Services is a commercial cloud service which allows users to rent a range of computers and other computing services. The following snippet shows how Parsl can be configured to provision nodes from the Elastic Compute Cloud (EC2) service. The first time this configuration is used, Parsl will configure a Virtual Private Cloud and other networking and security infrastructure that will be re-used in subsequent executions. The configuration uses the `AWSPProvider` to connect to AWS.

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

config = Config(
    executors=[
        HighThroughputExecutor(
            label='ec2_single_node',
            provider=AWSPProvider(
                # 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
                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',
            ),
        ),
    ]
)
```

(continues on next page)

(continued from previous page)

```

    1,
)

```

3.9.5 ASPIRE 1 (NSCC)

The following snippet shows an example configuration for accessing NSCC's **ASPIRE 1** supercomputer. This example uses the `HighThroughputExecutor` executor and connects to ASPIRE1's PBSPro scheduler. It also shows how `scheduler_options` parameter could be used for scheduling array jobs in PBSPro.

```

from parsl.providers import PBSProProvider
from parsl.launchers import MpiRunLauncher
from parsl.config import Config
from parsl.executors import HighThroughputExecutor
from parsl.addresses import address_by_interface
from parsl.monitoring.monitoring import MonitoringHub

config = Config(
    executors=[
        HighThroughputExecutor(
            label="htex",
            heartbeat_period=15,
            heartbeat_threshold=120,
            worker_debug=True,
            max_workers=4,
            address=address_by_interface('ib0'),
            provider=PBSProProvider(
                launcher=MpiRunLauncher(),
                # PBS directives (header lines): for array jobs pass '-J' option
                scheduler_options='#PBS -J 1-10',
                # Command to be run before starting a worker, such as:
                # 'module load Anaconda; source activate parsl_env'.
                worker_init='',
                # number of compute nodes allocated for each block
                nodes_per_block=3,
                min_blocks=3,
                max_blocks=5,
                cpus_per_node=24,
                # medium queue has a max walltime of 24 hrs
                walltime='24:00:00'
            ),
        ),
    ],
    monitoring=MonitoringHub(
        hub_address=address_by_interface('ib0'),
        hub_port=55055,
        resource_monitoring_interval=10,
    ),
    strategy='simple',
    retries=3,
    app_cache=True,
    checkpoint_mode='task_exit'
)

```

3.9.6 Blue Waters (NCSA)

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.launchers import AprunLauncher
from parsl.providers import TorqueProvider

config = Config(
    executors=[
        HighThroughputExecutor(
            label="bw_htex",
            cores_per_worker=1,
            worker_debug=False,
            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
                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'
            ),
        ),
    ],
)
```

3.9.7 Bridges (PSC)



The following snippet shows an example configuration for executing on the Bridges supercomputer at the Pitts-

burgh Supercomputing Center. 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.executors import HighThroughputExecutor

""" This config assumes that it is used to launch parsl tasks from the login nodes
of Bridges at PSC. Each job submitted to the scheduler will request 2 nodes for 10_
↳minutes.
"""

config = Config(
    executors=[
        HighThroughputExecutor(
            label='Bridges_HTEX_multinode',
            max_workers=1,
            provider=SlurmProvider(
                'YOUR_PARTITION_NAME', # Specify Partition / QOS, for eg. RM-small
                nodes_per_block=2,
                init_blocks=1,
                # string to prepend to #SBATCH blocks in the submit
                # script to the scheduler eg: '#SBATCH --gres=gpu:type:n'
                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(),
                walltime='00:10:00',
                # Slurm scheduler on Cori can be slow at times,
                # increase the command timeouts
                cmd_timeout=120,
            ),
        ],
    )
```


3.9.8 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

config = Config(
    executors=[
        HighThroughputExecutor(
            label='cc_in2p3_htex',
            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
            ),
        ),
    ],
)
```

3.9.9 CCL (Notre Dame, with Work Queue)

Work Queue



To utilize Work Queue with Parsl, please install the full CCTools software package within an appropriate Anaconda or Miniconda environment (instructions for installing Miniconda can be found [in the Conda install guide](#)):

```
$ conda create -y --name <environment> python=<version> conda-pack
$ conda activate <environment>
$ conda install -y -c conda-forge ndcctools parsl
```

This creates a Conda environment on your machine with all the necessary tools and setup needed to utilize Work Queue with the Parsl library.

The following snippet shows an example configuration for using the Work Queue distributed framework to run applications on remote machines at large. This examples uses the `WorkQueueExecutor` to schedule tasks locally, and assumes that Work Queue workers have been externally connected to the master using the `work_queue_factory` or `condor_submit_workers` command line utilities from CCTools. For more information on using Work Queue or to get help with running applications using CCTools, visit the [CCTools documentation online](#).

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

config = Config(
    executors=[
        WorkQueueExecutor(
            label="parsl_wq_example",
            port=9123,
            project_name="parsl_wq_example",
            shared_fs=False
        )
    ]
)
```


3.9.10 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.

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

config = Config(
    executors=[
        HighThroughputExecutor(
            label='Comet_HTEX_multinode',
            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,
            ),
        )
    ]
)
```

3.9.11 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.launchers import MpiRunLauncher
from parsl.providers import CobaltProvider
```

(continues on next page)

(continued from previous page)

```

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

```

3.9.12 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

```

(continues on next page)

(continued from previous page)

```

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=kn1,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,
            ),
        ),
    ],
)

```

3.9.13 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
from parsl.providers import SlurmProvider
from parsl.executors import HighThroughputExecutor
from parsl.launchers import SrunLauncher

""" 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",
            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(),
            ),
        ],
    )
```

3.9.14 Kubernetes Clusters



Kubernetes is an open-source system for container management, such as automating deployment and scaling of con-

ainers. 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',

                # 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,
            ),
        ],
)
```

3.9.15 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.executors import HighThroughputExecutor

config = Config(
    executors=[
        HighThroughputExecutor(
            label='Midway_HTEX_multinode',
            worker_debug=False,
            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=kn1,quad,cache'
                scheduler_options='',
                # Command to be run before starting a worker, such as:
                # 'module load Anaconda; source activate parsl_env'.
                worker_init='',
                walltime='00:30:00'
            ),
        ),
    ],
)
```

3.9.16 Open Science Grid



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. You will need to have a valid project name on the OSG. The configuration uses the *CondorProvider* to interface with the scheduler.

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

config = Config(
    executors=[
        HighThroughputExecutor(
            label='OSG_HTEX',
            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="""
+ProjectName = "MyProject"
Requirements = HAS_MODULES=?=TRUE
""",
                # Command to be run before starting a worker, such as:
                # 'module load Anaconda; source activate parsl_env'.
```

(continues on next page)

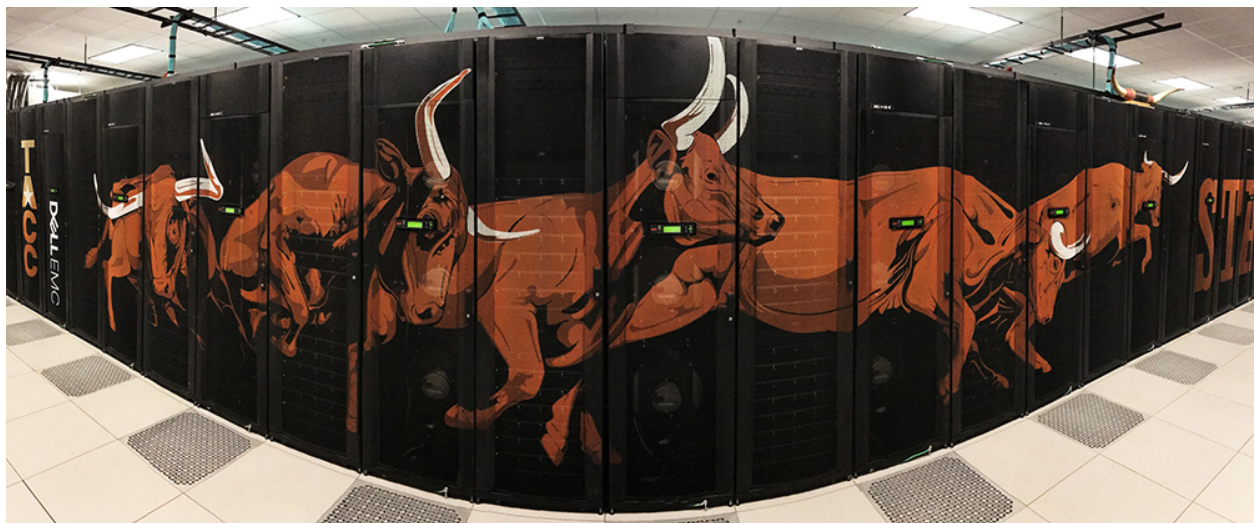
(continued from previous page)

```

        worker_init='''unset HOME; unset PYTHONPATH; module load python/3.7.0;
python3 -m venv parsl_env; source parsl_env/bin/activate; python3 -m pip install parsl
→'''
        walltime="00:20:00",
    ),
    worker_logdir_root='$OSG_WN_TMP',
    worker_ports=(31000, 31001)
)
]
)

```

3.9.17 Stampede2 (TACC)



The following snippet shows an example configuration for accessing TACC's **Stampede2** supercomputer. This example uses the `HighThroughput` 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.data_provider.globus import GlobusStaging

```

```

config = Config(
    executors=[
        HighThroughputExecutor(
            label='Stampede2_HTEX',
            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=kn1,quad,cache'

```

(continues on next page)

(continued from previous page)

```

        scheduler_options='',
        # 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='/'
    )]
)
],
)

```

3.9.18 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 *Jsruncher* to launch workers across the compute nodes.

```

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

from parsl.launchers import Jsruncher
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=Jsruncher(),
                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
            ),
        ],
    )

```

(continues on next page)

(continued from previous page)

```

    ],
)

```

3.9.19 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
from parsl.executors import HighThroughputExecutor

config = Config(
    executors=[
        HighThroughputExecutor(
            label='theta_local_htex_multinode',
            max_workers=4,
            provider=CobaltProvider(
                queue='YOUR_QUEUE',
                account='YOUR_ACCOUNT',

```

(continues on next page)

(continued from previous page)

```

        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,
    ),
),
1,
)

```

3.9.20 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)

```

3.10 Monitoring

Parsl includes a monitoring system to capture task state as well as resource usage over time. The Parsl monitoring system aims to provide detailed information and diagnostic capabilities to help track the state of your programs, down to the individual apps that are executed on remote machines.

The monitoring system records information to an SQLite database while a workflow runs. This information can then be visualised in a web dashboard using the `parsl-visualize` tool, or queried using SQL using regular SQLite tools.

3.10.1 Monitoring configuration

Parsl monitoring is only supported with the [HighThroughputExecutor](#).

The following example shows how to enable monitoring in the Parsl configuration. Here the [MonitoringHub](#) is specified to use port 55055 to receive monitoring messages from workers every 10 seconds.

```

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

```

(continues on next page)

(continued from previous page)

```

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
)

```

3.10.2 Visualization

To run the web dashboard utility `parsl-visualize` you first need to install its dependencies:

```
$ pip install parsl[monitoring]
```

To view the web dashboard while or after a Parsl program has executed, run the `parsl-visualize` utility:

```
$ parsl-visualize
```

By default, this command expects that the default `monitoring.db` database is used in the current working directory. Other databases can be loaded by passing the database URI on the command line. For example, if the full path to the database is `/tmp/my_monitoring.db`, run:

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

By default, the visualization web server listens on `127.0.0.1:8080`. If the web server is deployed on a machine with a web browser, the dashboard can be accessed in the browser at `127.0.0.1:8080`. If the web server is deployed on a remote machine, such as the login node of a cluster, you will need to use an ssh tunnel from your local machine to the cluster:

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


This command will bind your local machine's port 50000 to the remote cluster's port 8080. The dashboard can then be accessed via the local machine's browser at `127.0.0.1:50000`.

Warning: Alternatively you can deploy the visualization server on a public interface. However, first check that this is allowed by the cluster's security policy. The following example shows how to deploy the web server on a public port (i.e., open to Internet via `public_IP:55555`):

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

Workflows Page

The workflows page lists all Parsl workflows that have been executed with monitoring enabled with the selected database. It provides a high level summary of workflow state as shown below:

Parsl

Workflows Documentation


Workflows

Name	Version	Owner	Status	Runtime (s)	Tasks	Actions
test_fan_inout.py	2019-06-13 10:58:14	yadu	Completed	0:00:25	<div><div>21</div><div>0</div></div>	View
test_monitor_on_fail.py	2019-06-13 11:02:02	yadu	Completed	0:00:02	<div><div>0</div><div>1</div></div>	View

Throughout the dashboard, all blue elements are clickable. For example, clicking a specific workflow name from the table takes you to the Workflow Summary page described in the next section.

Workflow Summary

The workflow summary page captures the run level details of a workflow, including 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 invocation count for each.

 Parsl

Workflows

Documentation

test_fan_inout.py

Workflow Summary

- **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

App Summary

Name	Count
add_inc	5
inc	16

[View workflow DAG -- colors grouped by apps](#)

[View workflow DAG -- colors grouped by task states](#)

[View workflow resource usage](#)

The workflow summary also presents three different views of the workflow:

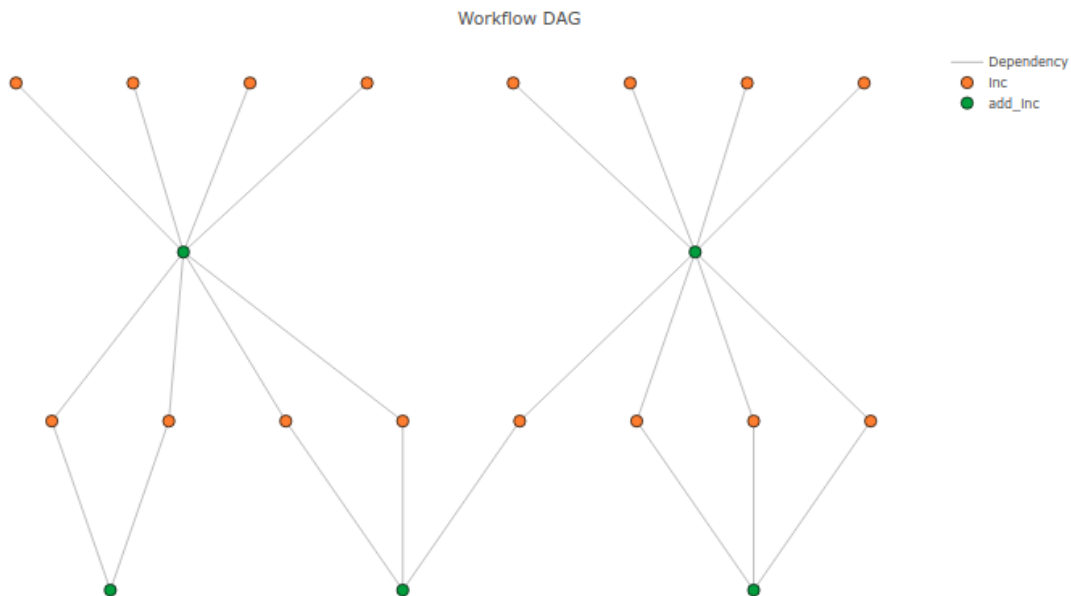
- Workflow DAG - with apps differentiated by colors: This visualization is useful to visually inspect the dependency structure of the workflow. Hovering over the nodes in the DAG shows a tooltip for the app represented by the node and it's task ID.

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](#)



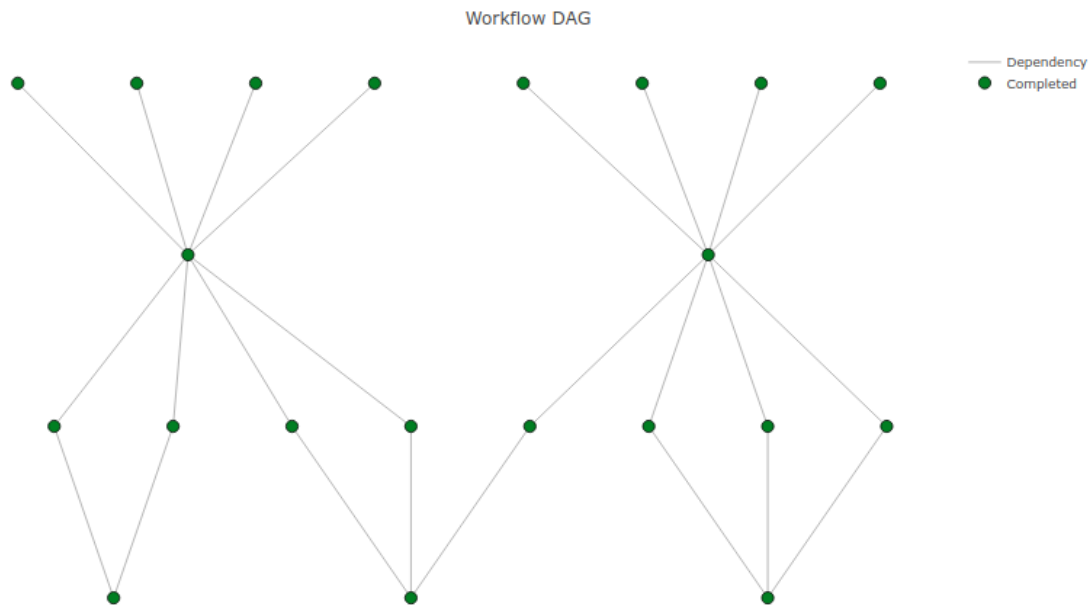
- Workflow DAG - with task states differentiated by colors: This visualization is useful to identify what tasks have been completed, failed, or are currently pending.

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 example, cumulative CPU/Memory utilization across workers over time.

test_fan_inout.py

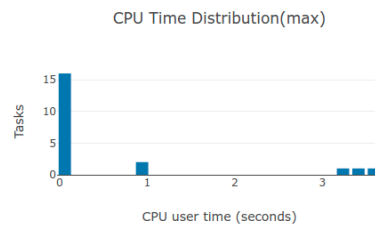
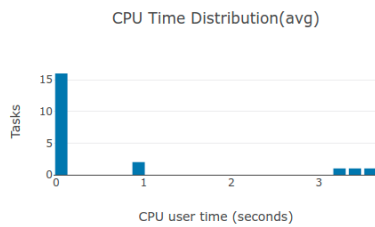
- **Started:** 2019-06-13 10:58:14
- **Completed:** 2019-06-13 10:58:39
- **Workflow duration:** 0:00:25
- **Owner:** yadu
- **host:** borgmachine2
- **rundir:** /home/yadu/src/parsl/parsl/tests/manual_tests/runinfo/000
- **tasks_failed_count:** 0
- **tasks_completed_count:** 21

[View workflow DAG – colors grouped by apps](#)

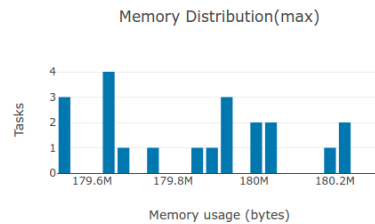
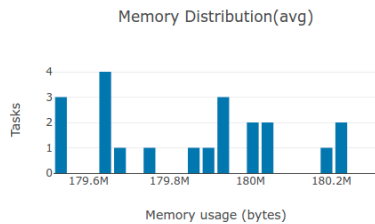
[View workflow DAG – colors grouped by task states](#)

[View workflow task summary](#)

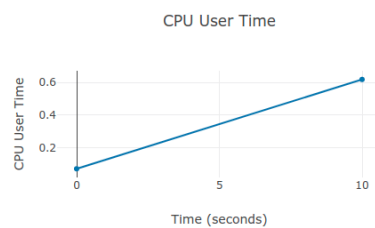
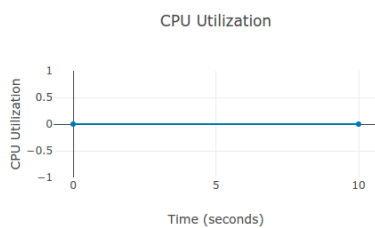
CPU Usage



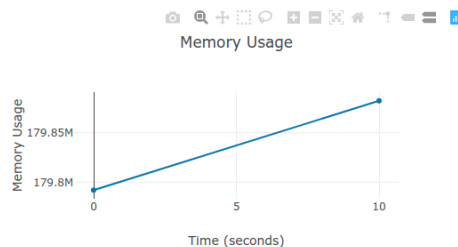
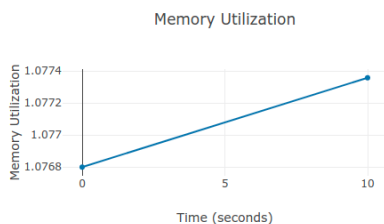
Memory Usage



CPU usage



Memory usage



3.11 Example parallel patterns

Parsl can be used to implement a wide range of parallel programming patterns, from bag of tasks through to nested workflows. Parsl implicitly assembles a dataflow dependency graph based on the data shared between apps. The flexibility of this model allows for the implementation of a wide range of parallel programming and workflow patterns.

Parsl is also designed to address broad execution requirements, from programs that run many short tasks to those that run a few long tasks.

Below we illustrate a range of parallel programming and workflow patterns. It is important to note that this set of examples is by no means comprehensive.

3.11.1 Bag of Tasks

Parsl can be used to execute a large bag of tasks. In this case, Parsl assembles the set of tasks (represented as Parsl apps) and manages their concurrent execution on available resources.

```
from parsl import python_app

parsl.load()

# Map function that returns double the input integer
@python_app
def app_random():
    import random
    return random.random()

results = []
for i in range(0, 10):
    x = app_random()
    mapped_results.append(x)

for r in results:
    print(r.result())
```

3.11.2 Sequential workflows

Sequential workflows can be created by passing an AppFuture from one task to another. For example, in the following program the `generate` app (a Python app) generates a random number that is consumed by the `save` app (a Bash app), which writes it to a file. Because `save` cannot execute until it receives the message produced by `generate`, the two apps execute in sequence.

```
from parsl import python_app

parsl.load()

# 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
```

(continues on next page)

(continued from previous page)

```
@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())
```

3.11.3 Parallel workflows

Parallel execution occurs automatically in Parsl, respecting dependencies among app executions. In the following example, three instances of the `wait_sleep_double` app are created. The first two execute concurrently, as they have no dependencies; the third must wait until the first two complete and thus the `doubled_x` and `doubled_y` futures have values. Note that this sequencing occurs even though `wait_sleep_double` does not in fact use its second and third arguments.

```
from parsl import python_app

parsl.load()

@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 app depends on the first two:
#   doubled_x   doubled_y   (2 s)
#       \       /
#       doubled_x_z   (2 s)
doubled_z = wait_sleep_double(10, doubled_x, doubled_y)

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

3.11.4 Parallel workflows with loops

A common approach to executing Parsl apps in parallel is via loops. The following example uses a loop to create many random numbers in parallel.

```
from parsl import python_app

parsl.load()

@python_app
```

(continues on next page)

(continued from previous page)

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

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

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

In the preceding example, the execution of different tasks is coordinated by passing Python objects from producers to consumers. In other cases, it can be convenient to pass data in files, as in the following reformulation. Here, 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.

```
from parsl import python_app, bash_app

parsl.load()

@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())
```

3.11.5 MapReduce

MapReduce is a common pattern used in data analytics. It is composed of a map phase that filters values and a reduce phase that aggregates values. The following example demonstrates how Parsl can be used to specify a MapReduce computation in which the map phase doubles a set of input integers and the reduce phase computes the sum of those results.

```
from parsl import python_app

parsl.load()

# 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)

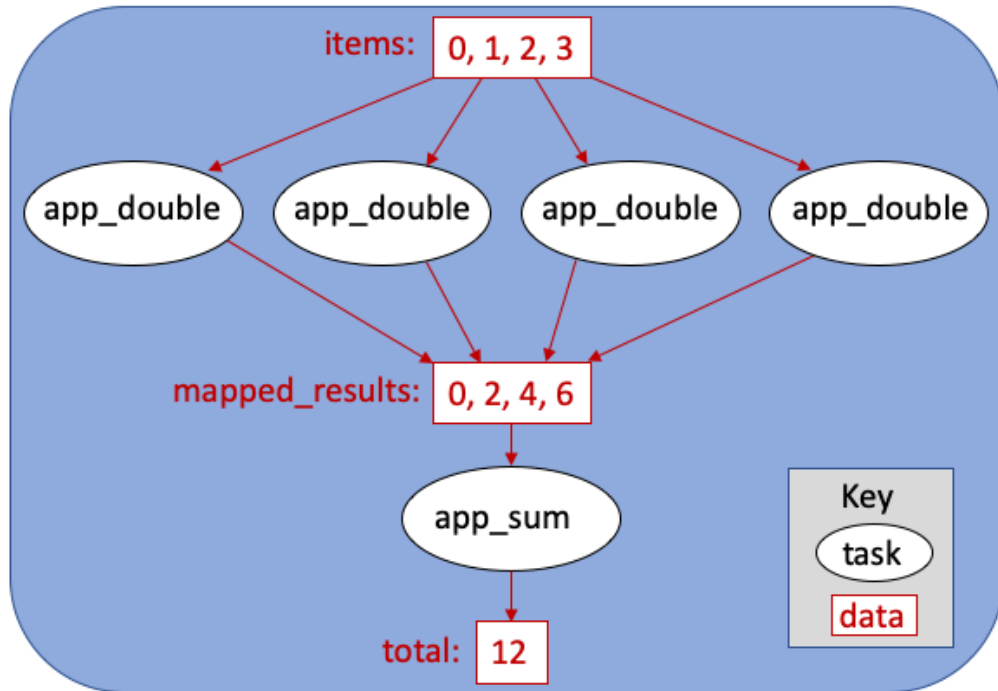
# Create a list of integers
items = range(0,4)

# Map phase: apply the double *app* function to each item in list
mapped_results = []
for i in items:
    x = app_double(i)
    mapped_results.append(x)

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

print(total.result())
```

The program first defines two Parsl apps, `app_double` and `app_sum`. It then makes calls to the `app_double` app with a set of input values. It then passes the results from `app_double` to the `app_sum` app to aggregate values into a single result. These tasks execute concurrently, synchronized by the `mapped_results` variable. The following figure shows the resulting task graph.



3.12 Structuring Parsl programs

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

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

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.

```

from parsl.config import Config
from parsl.channels import LocalChannel
from parsl.executors import HighThroughputExecutor
from parsl.providers import LocalProvider

htex_config = Config(
    executors=[
        HighThroughputExecutor(
            label="htex_local",
            cores_per_worker=1,
            provider=LocalProvider(
                channel=LocalChannel(),
            ),
        ),
    ],
)

```

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

```
import parsl
from config import htex_config
from library import increment

parsl.load(htex_config)

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
```

3.13 Join Apps

Join apps allows an app to define a sub-workflow: the app can launch other apps and incorporate them into the main task graph. They can be specified using the `join_app` decorator.

Join apps allow more nuanced dependencies to be expressed that can help with:

- increased concurrency - helping with strong scaling
- more focused error propagation - allowing more of an ultimately failing workflow to complete
- more useful monitoring information

3.13.1 Usage

A `join_app` looks quite like a `python_app`, but should return a `Future`, rather than a value. After the python code has run, the app invocation will not complete until that future has completed, and the return value of the `join_app` will be the return value (or exception) from the returned future.

For example:

```
@python_app
def some_app():
    return 3

@join_app
```

(continues on next page)

(continued from previous page)

```
def example():
    x: Future = some_app()
    return x  # note that x is a Future, not a value

# example.result() == 3
```

3.13.2 What/why/how can you do with a join app

join apps are useful when a workflow needs to launch some apps, but it doesn't know what those apps are until some earlier apps are completed.

For example, a pre-processing stage might be followed by n middle stages, but the value of n is not known until pre-processing is complete; or the choice of app to run might depend on the output of pre-processing.

In the following example, a pre-processing stage is followed by a choice of option 1 or option 2 apps, with a post-processing stage afterwards. All of the example apps are toy apps that are intended to demonstrate control/data flow but they are based on a real use case.

Here is the implementation using join apps. Afterwards, there are some examples of the problems that arise trying to implement this without join apps.

```
@python_app
def pre_process():
    return 3

@python_app
def option_one(x):
    # do some stuff
    return x*2

@python_app
def option_two(x):
    # do some more stuff
    return (-x) * 2

@join_app
def process(x):
    if x > 0:
        return option_one(x)
    else:
        return option_two(x)

@python_app
def post_process(x):
    return str(x) # convert x to a string

# here is a simple workflow using these apps:
# post_process(process(pre_process())).result() == "6"
# pre_process gives the number 3, process turns it into 6,
# and post_process stringifys it to "6"
```

So why do we need process to be a @join_app for this to work?

- Why can't process be a regular python function?

process needs to inspect the value of x to make a decision about what app to launch. So it needs to defer execution until after the pre-processing stage has completed. In parsl, the way to defer that is using apps: the execution of

process will happen when the future returned by `pre_process` has completed.

- Why can't process be a `@python_app`?

A python app, if run in a `ThreadPoolExecutor`, can launch more parsl apps; so a python app implementation of `process()` would be able to inspect `x` and launch `option_{one, two}`.

From launching the `option_{one, two}` app, the app body python code would get a `Future[int]` - a Future that will eventually contain `int`.

But now, we want to (at submission time) invoke `post_process`, and have it wait until the relevant `option_{one, two}` app has completed.

If we don't have join apps, how can we do this?

We could make process wait for `option_{one, two}` to complete, before returning, like this:

```
@python_app
def process(x):
    if x > 0:
        f = option_one(x)
    else:
        f = option_two(x)
    return f.result()
```

but this will block the worker running process until `option_{one, two}` has completed. If there aren't enough workers to run `option_{one, two}` this can even deadlock. (principle: apps should not wait on completion of other apps and should always allow parsl to handle this through dependencies)

We could make process return the Future to the main workflow thread:

```
@python_app
def process(x):
    if x > 0:
        f = option_one(x)
    else:
        f = option_two(x)
    return f  # f is a Future[int]

# process(3) is a Future[Future[int]]
```

What comes out of invoking `process(x)` now is a nested `Future[Future[int]]` - it's a promise that eventually process will give you a promise (from `option_{one, two}`) that will eventually give you an `int`.

We can't pass that future into `post_process`... because `post_process` wants the final `int`, and that future will complete before the `int` is ready, and that (outer) future will have as its value the inner future (which won't be complete yet).

So we could wait for the result in the main workflow thread:

```
f_outer = process(pre_process())  # Future[Future[int]]
f_inner = f_outer.result()  # Future[int]
result = post_process(f_inner)
# result == "6"
```

But this now blocks the main workflow thread. If we really only need to run these three lines, that's fine, but what about if we are in a for loop that sets up 1000 parametrised iterations:

```
for x in [1..1000]:
    f_outer = process(pre_process(x))  # Future[Future[int]]
    f_inner = f_outer.result()  # Future[int]
    result = post_process(f_inner)
```


The `for` loop can only iterate after `pre_processing` is done for each iteration - it is unnecessarily serialised by the `.result()` call, so that `pre_processing` cannot run in parallel.

So, the rule about not calling `.result()` applies in the main workflow thread too.

What join apps add is the ability for `parsl` to unwrap that `Future[Future[int]]` into a `Future[int]` in a “sensible” way (eg it doesn’t need to block a worker).

3.13.3 Terminology

The term “join” comes from use of monads in functional programming, especially Haskell.

3.14 Performance and Scalability

Parsl is designed to scale from small to large systems .

3.14.1 Scalability

We studied strong and weak scaling on the Blue Waters supercomputer. In strong scaling, the total problem size is fixed; in weak scaling, the problem size per CPU core is fixed. In both cases, we measure completion time as a function of number of CPU cores. An ideal framework should scale linearly, which for strong scaling means that speedup scales with the number of cores, and for weak scaling means that completion time remains constant as the number of cores increases.

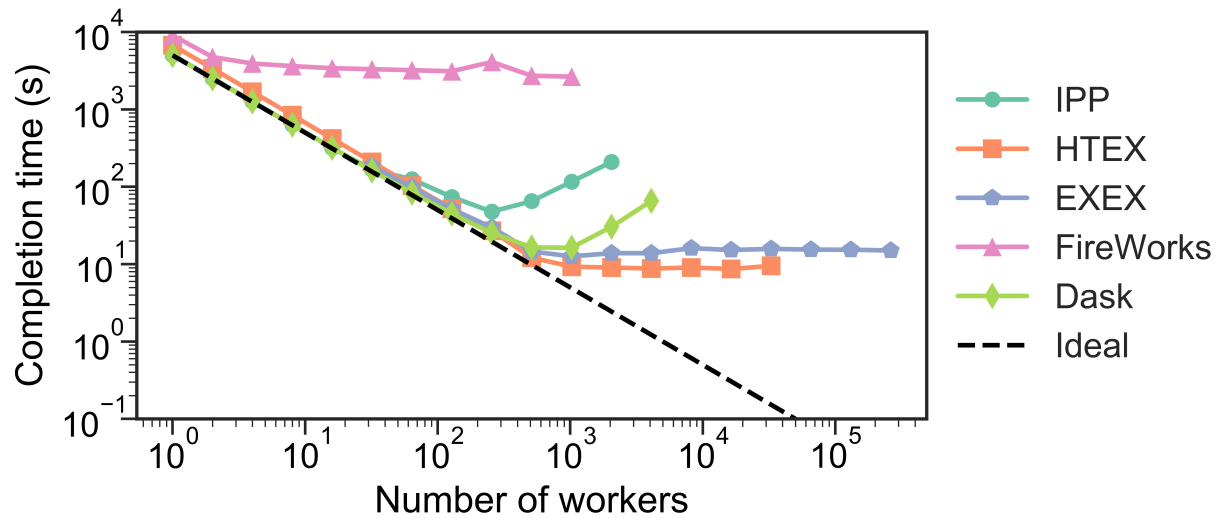
To measure the strong and weak scaling of Parsl executors, we created Parsl programs to run tasks with different durations, ranging from a “no-op”—a Python function that exits immediately—to tasks that sleep for 10, 100, and 1,000 ms. For each executor we deployed a worker per core on each node.

While we compare here with IPP, Fireworks, and Dask Distributed, we note that these systems are not necessarily designed for Parsl-like workloads or scale.

Further results are presented in our [HPDC paper](#).

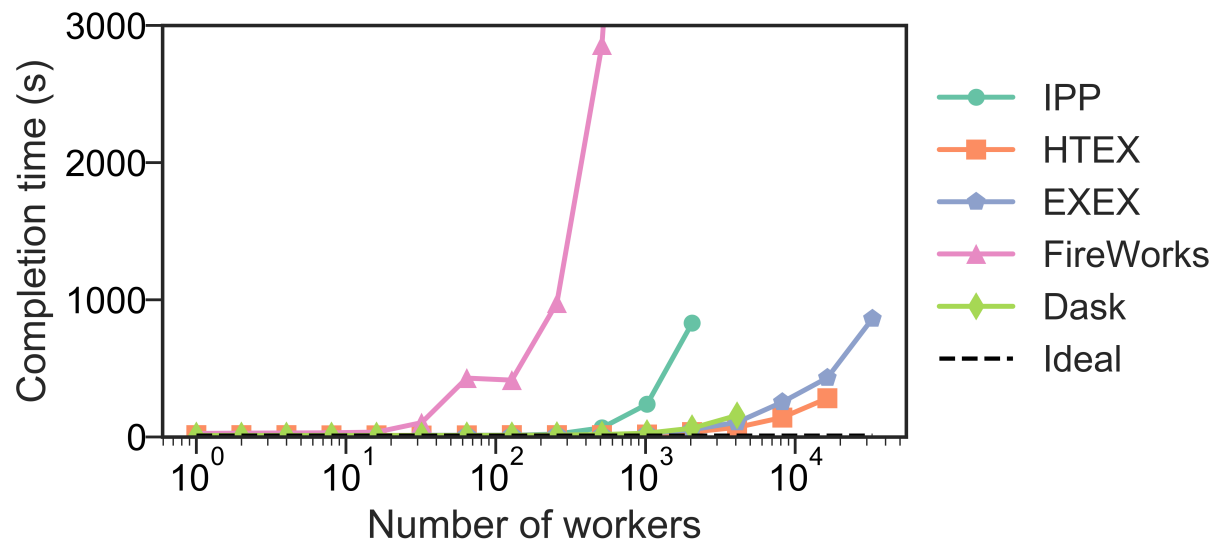
Strong scaling

The figures below show the strong scaling results for 5,000 1-second sleep tasks. HTEX provides good performance in all cases, slightly exceeding what is possible with EXEX, while EXEX scales to significantly more workers than the other executors and frameworks. Both HTEX and EXEX remain nearly constant, indicating that they likely will continue to perform well at larger scales.



Weak scaling

Here, we launched 10 tasks per worker, while increasing the number of workers. (We limited experiments to 10 tasks per worker, as on 3,125 nodes, that represents $3,125 \text{ nodes} \times 32 \text{ workers/node} \times 10 \text{ tasks/worker}$, or 1M tasks.) The figure below shows our results. We observe that HTEX and EXEX outperform other executors and frameworks with more than 4,096 workers (128 nodes). All frameworks exhibit similar trends, with completion time remaining close to constant initially and increasing rapidly as the number of workers increases.



3.14.2 Throughput

We measured the maximum throughput of all the Parsl executors, on the UChicago Research Computing Center’s Midway Cluster. To do so, we ran 50,000 “no-op” tasks on a varying number of workers and recorded the completion times. The throughput is computed as the number of tasks divided by the completion time. HTEX, and EXEX achieved maximum throughputs of 1,181 and 1,176 tasks/s, respectively.

3.14.3 Summary

The table below summarizes the scale at which we have tested Parsl executors. The maximum number of nodes and workers for HTEX and EXEX is limited by the size of allocation available during testing on Blue Waters. The throughput results are collected on Midway.

Executor	Max # workers	Max # nodes	Max tasks/second
IPP	2,048	64	330
HTEX	65,536	2,048	1,181
EXEX	262,144	8,192	1,176

3.15 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.

3.15.1 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, it is important that we provide aggregate 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).

3.15.2 Opt-In

We have chosen opt-in collection rather than opt-out with the hope that developers and researchers will choose to send us this information. The reason is that we need this data - it is a requirement for funding.

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`*).

3.15.3 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 and Python version
- OS and OS version

3.15.4 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.

3.15.5 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.

3.15.6 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 tasks complete successfully?
- Of the tasks that fail, what is the most common fault code returned?

3.15.7 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.

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 `pickleable` 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 *HighThroughputExecutor* 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 HighThroughputExecutor
from parsl.addresses import address_by_route, address_by_query, address_by_hostname

config = Config(
    executors=[
        HighThroughputExecutor(
            label='ALCF_theta_local',
            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: 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: [Configuration](#).

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
→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 in [issue 59](#) 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 configuration:

```
from libsubmit.providers import Cobalt
from parsl.config import Config
from parsl.executors import HighThroughputExecutor

config = Config(
    executors=[
        HighThroughputExecutor(
            label='ALCF_theta_local',
            provider=Cobalt(),
            worker_port_range=(50000, 55000),
            interchange_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 [these instructions](#).

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 *in the quickstart guide*) will automatically be added to the list of kernels.

4.15 Addressing SerializationError

As of v1.0.0, Parsl will raise a `SerializationError` when it encounters an object that Parsl cannot serialize. This applies to objects passed as arguments to an app, as well as objects returned from the app.

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`. For example the following data types are known to have issues with serializability :

- Closures
- Objects of complex classes with no `__dict__` or `__getstate__` methods defined
- System objects such as file descriptors, sockets and locks (e.g `threading.Lock`)

If Parsl raises a `SerializationError`, first identify what objects are problematic with a quick test:

```
import pickle
# If non-serializable you will get a TypeError
pickle.dumps(YOUR_DATA_OBJECT)
```

If the data object simply is complex, please refer [here](#) for more details on adding custom mechanisms for supporting serialization.

4.16 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
```

(continues on next page)

(continued from previous page)

```
        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}  
}
```


API REFERENCE GUIDE

5.1 Core

<code>parsl.app.app.python_app</code>	Decorator function for making python apps.
<code>parsl.app.app.bash_app</code>	Decorator function for making bash apps.
<code>parsl.app.app.join_app</code>	
<code>parsl.dataflow.futures.AppFuture</code>	An AppFuture wraps a sequence of Futures which may fail and be retried.
<code>parsl.dataflow.dflow.DataFlowKernelLoader</code>	Manage which DataFlowKernel is active.
<code>parsl.monitoring.MonitoringHub</code>	

5.1.1 `parsl.app.app.python_app`

`parsl.app.app.python_app` (*function=None*, *data_flow_kernel: Optional[`parsl.dataflow.dflow.DataFlowKernel`] = None*, *cache: bool = False*, *executors: Union[List[str], typing_extensions.Literal[all]] = 'all'*, *ignore_for_cache: Optional[List[str]] = None*, *join: bool = False*)

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.
- **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.
- **join** (*bool*) – If True, this app will be a join app: the decorated python code must return a Future (rather than a regular value), and the corresponding AppFuture will complete when that inner future completes.

5.1.2 parsl.app.app.bash_app

```
parsl.app.app.bash_app (function=None, data_flow_kernel: Optional[parsl.dataflow.dflow.DataFlowKernel] = None, cache: bool = False, executors: Union[List[str], typing_extensions.Literal[all]] = 'all', ignore_for_cache: Optional[List[str]] = None)
```

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.

5.1.3 parsl.app.app.join_app

```
parsl.app.app.join_app (function=None, data_flow_kernel: Optional[parsl.dataflow.dflow.DataFlowKernel] = None, cache: bool = False, ignore_for_cache: Optional[List[str]] = None)
```

5.1.4 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 as 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

<code>__init__(task_def)</code>	Initialize the AppFuture.
<code>add_done_callback(fn)</code>	Attaches a callable that will be called when the future finishes.
<code>cancel()</code>	Cancel the future if possible.
<code>cancelled()</code>	Return True if the future was cancelled.
<code>done()</code>	Return True if the future was cancelled or finished executing.
<code>exception([timeout])</code>	Return the exception raised by the call that the future represents.
<code>result([timeout])</code>	Return the result of the call that the future represents.
<code>running()</code>	Return True if the future is currently executing.
<code>set_exception(exception)</code>	Sets the result of the future as being the given exception.
<code>set_result(result)</code>	Sets the return value of work associated with the future.
<code>set_running_or_notify_cancel()</code>	Mark the future as running or process any cancel notifications.
<code>task_status()</code>	Returns the status of the task that will provide the value for this future.

Attributes

<code>outputs</code>
<code>stderr</code>
<code>stdout</code>
<code>tid</code>

5.1.5 `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

<code>__init__()</code>	Initialize self.
<code>clear()</code>	Clear the active DataFlowKernel so that a new one can be loaded.
<code>dfk()</code>	Return the currently-loaded DataFlowKernel.
<code>load([config])</code>	Load a DataFlowKernel.
<code>wait_for_current_tasks()</code>	Waits for all tasks in the task list to be completed, by waiting for their AppFuture to be completed.

5.1.6 `parsl.monitoring.MonitoringHub`

```
class parsl.monitoring.MonitoringHub (hub_address: str, hub_port: Optional[int] = None,
                                     hub_port_range: Tuple[int, int] = (55050, 56000),
                                     client_address: str = '127.0.0.1', client_port_range:
                                     Tuple[int, int] = (55000, 56000), workflow_name: Op-
                                     tional[str] = None, workflow_version: Optional[str] =
                                     None, logging_endpoint: str = 'sqlite:///monitoring.db',
                                     logdir: Optional[str] = None, monitoring_debug: bool
                                     = False, resource_monitoring_enabled: bool = True,
                                     resource_monitoring_interval: float = 30)
```

```
__init__ (hub_address: str, hub_port: Optional[int] = None, hub_port_range: Tuple[int, int]
          = (55050, 56000), client_address: str = '127.0.0.1', client_port_range: Tuple[int, int]
          = (55000, 56000), workflow_name: Optional[str] = None, workflow_version: Op-
          tional[str] = None, logging_endpoint: str = 'sqlite:///monitoring.db', logdir: Optional[str]
          = None, monitoring_debug: bool = False, resource_monitoring_enabled: bool = True, re-
          source_monitoring_interval: float = 30)
```

Parameters

- **hub_address** (*str*) – The ip address at which the workers will be able to reach the Hub.
- **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. Default: (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. Default: (55000, 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, host-name, 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** (*float*) – The time interval, in seconds, at which the monitoring records the resource usage of each task. Default: 30 seconds

Methods

<code>__init__(hub_address[, hub_port, ...])</code>	param hub_address The ip address at which the workers will be able to reach the Hub.
<code>close()</code>	
<code>monitor_wrapper(f, try_id, task_id, ...)</code>	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.
<code>send(mtype, message)</code>	
<code>start(run_id)</code>	

5.2 Configuration

<code>parsl.config.Config</code>	Specification of Parsl configuration options.
<code>parsl.set_stream_logger</code>	Add a stream log handler.
<code>parsl.set_file_logger</code>	Add a stream log handler.
<code>parsl.addresses.address_by_hostname</code>	Returns the hostname of the local host.
<code>parsl.addresses.address_by_interface</code>	Returns the IP address of the given interface name, e.g.
<code>parsl.addresses.address_by_query</code>	Finds an address for the local host by querying ipify.
<code>parsl.addresses.address_by_route</code>	Finds an address for the local host by querying the local routing table for the route to Google DNS.
<code>parsl.utils.get_all_checkpoints</code>	Finds the checkpoints from all runs in the rundir.
<code>parsl.utils.get_last_checkpoint</code>	Finds the checkpoint from the last run, if one exists.

5.2.1 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,
                          garbage_collect: bool = True, internal_tasks_max_threads: int = 10,
                          retries: int = 0, run_dir: str = 'run-info', 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 *ParslExecutor* instances to use for executing tasks. 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. See `parsl.utils.get_all_checkpoints()` and `parsl.utils.get_last_checkpoint()` for helpers. 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'`.
- **garbage_collect** (*bool, optional*) – Delete task records from DFK when tasks have completed. Default: `True`
- **internal_tasks_max_threads** (*int, optional*) – Maximum number of threads to allocate for submit side internal tasks such as some data transfers or `@joinapps`. Default is 10.
- **monitoring** (*MonitoringHub, optional*) – The config to use for database monitoring. Default is `None` which does not log to a database.
- **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 allowed for an executor before strategy could shut down unused resources (scheduler jobs). 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, garbage_collect: bool = True, internal_tasks_max_threads: int = 10, 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

<code>__init__</code> ([executors, app_cache, ...])	Initialize self.
---	------------------

Attributes

<code>executors</code>

5.2.2 `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

5.2.3 `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

5.2.4 `parsl.addresses.address_by_hostname`

`parsl.addresses.address_by_hostname` () → *str*

Returns the hostname of the local host.

This will return an unusable value when the hostname cannot be resolved from workers.

5.2.5 `parsl.addresses.address_by_interface`

`parsl.addresses.address_by_interface(ifname: str) → str`
 Returns the IP address of the given interface name, e.g. 'eth0'

This is taken from a Stack Overflow answer: <https://stackoverflow.com/questions/24196932/how-can-i-get-the-ip-address-of-eth0-in-python#24196955>

Parameters `ifname` (*str*) – Name of the interface whose address is to be returned. Required.

5.2.6 `parsl.addresses.address_by_query`

`parsl.addresses.address_by_query(timeout: float = 30) → str`
 Finds an address for the local host by querying ipify. This may return an unusable value when the host is behind NAT, or when the internet-facing address is not reachable from workers. Parameters: _____

timeout [float] Timeout for the request in seconds. Default: 30s

5.2.7 `parsl.addresses.address_by_route`

`parsl.addresses.address_by_route() → str`
 Finds an address for the local host by querying the local routing table for the route to Google DNS.
 This will return an unusable value when the internet-facing address is not reachable from workers.

5.2.8 `parsl.utils.get_all_checkpoints`

`parsl.utils.get_all_checkpoints(rundir: str = 'runinfo') → List[str]`
 Finds the checkpoints from all runs in the rundir.

Kwargs:

- `rundir(str)` : Path to the runinfo directory

Returns

- a list suitable for the `checkpoint_files` parameter of *Config*

5.2.9 `parsl.utils.get_last_checkpoint`

`parsl.utils.get_last_checkpoint(rundir: str = 'runinfo') → List[str]`
 Finds the checkpoint from the last run, if one exists.

Note that checkpoints are incremental, and this helper will not find previous checkpoints from earlier than the most recent run. If you want that behaviour, see *get_all_checkpoints*.

Kwargs:

- `rundir(str)` : Path to the runinfo directory

Returns

- a list suitable for the `checkpoint_files` parameter of *Config*, with 0 or 1 elements

5.3 Channels

<code>parsl.channels.base.Channel</code>	For certain resources such as campus clusters or supercomputers at research laboratories, resource requirements may require authentication.
<code>parsl.channels.LocalChannel</code>	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
<code>parsl.channels.SSHChannel</code>	SSH persistent channel.
<code>parsl.channels.OAuthSSHChannel</code>	SSH persistent channel.
<code>parsl.channels.SSHInteractiveLoginChannel</code>	SSH persistent channel.

5.3.1 `parsl.channels.base.Channel`

class `parsl.channels.base.Channel`

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.

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)	<-----	
src, dst_dir	----->	push_file
dst_path	<-----	
dst_script_dir	<-----	script_dir
	+-----	

Channels should ensure that each launched command runs in a new process group, so that providers (such as *AdHocProvider* and *LocalProvider*) which terminate long running commands using process groups can do so.

__init__ ()

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

Methods

<code>__init__()</code>	Initialize self.
<code>abspath(path)</code>	Return the absolute path.
<code>close()</code>	Closes the channel.
<code>execute_wait(cmd[, walltime, envs])</code>	Executes the cmd, with a defined walltime.
<code>isdir(path)</code>	Return true if the path refers to an existing directory.
<code>makedirs(path[, mode, exist_ok])</code>	Create a directory.
<code>pull_file(remote_source, local_dir)</code>	Transport file on the remote side to a local directory
<code>push_file(source, dest_dir)</code>	Channel will take care of moving the file from source to the destination directory

Attributes

<code>script_dir</code>	This is a property.
-------------------------	---------------------

5.3.2 parsl.channels.LocalChannel

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

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*)

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

<code>__init__([userhome, envs, script_dir])</code>	Initialize the local channel.
<code>abspath(path)</code>	Return the absolute path.
<code>close()</code>	There's nothing to close here, and this really doesn't do anything
<code>execute_wait(cmd[, walltime, envs])</code>	Synchronously execute a commandline string on the shell.
<code>isdir(path)</code>	Return true if the path refers to an existing directory.
<code>makedirs(path[, mode, exist_ok])</code>	Create a directory.
<code>pull_file(remote_source, local_dir)</code>	Transport file on the remote side to a local directory
<code>push_file(source, dest_dir)</code>	If the source files dirpath is the same as <code>dest_dir</code> , a copy is not necessary, and nothing is done.

Attributes

<code>script_dir</code>	This is a property.
-------------------------	---------------------

5.3.3 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,          key_filename=None,
                                host_keys_filename=None)
```

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,          key_filename=None,
          host_keys_filename=None)
```

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
- `key_filename` (string or list): the filename, or list of filenames, of optional private key(s)

Raises:

Methods

<code>__init__</code> (hostname[, username, password, ...])	Initialize a persistent connection to the remote system.
<code>abspath</code> (path)	Return the absolute path on the remote side.
<code>close</code> ()	Closes the channel.
<code>execute_wait</code> (cmd[, walltime, envs])	Synchronously execute a commandline string on the shell.
<code>isdir</code> (path)	Return true if the path refers to an existing directory.
<code>makedirs</code> (path[, mode, exist_ok])	Create a directory on the remote side.
<code>prepend_envs</code> (cmd[, env])	
<code>pull_file</code> (remote_source, local_dir)	Transport file on the remote side to a local directory

continues on next page

Table 14 – continued from previous page

<code>push_file(local_source, remote_dir)</code>	Transport a local file to a directory on a remote machine
--	---

Attributes

<code>script_dir</code>	This is a property.
-------------------------	---------------------

5.3.4 `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

<code>__init__</code> (hostname[, username, script_dir, ...])	Initialize a persistent connection to the remote system.
<code>abspath</code> (path)	Return the absolute path on the remote side.
<code>close</code> ()	Closes the channel.
<code>execute_wait</code> (cmd[, walltime, envs])	Synchronously execute a commandline string on the shell.
<code>isdir</code> (path)	Return true if the path refers to an existing directory.
<code>makedirs</code> (path[, mode, exist_ok])	Create a directory on the remote side.
<code>prepend_envs</code> (cmd[, env])	
<code>pull_file</code> (remote_source, local_dir)	Transport file on the remote side to a local directory
<code>push_file</code> (local_source, remote_dir)	Transport a local file to a directory on a remote machine

Attributes

<code>script_dir</code>	This is a property.
-------------------------	---------------------

5.3.5 parsl.channels.SSHInteractiveLoginChannel

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

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*)

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

<code>__init__</code> (<i>hostname</i> [, <i>username</i> , <i>password</i> , ...])	Initialize a persistent connection to the remote system.
<code>abspath</code> (<i>path</i>)	Return the absolute path on the remote side.
<code>close</code> ()	Closes the channel.
<code>execute_wait</code> (<i>cmd</i> [, <i>walltime</i> , <i>envs</i>])	Synchronously execute a commandline string on the shell.
<code>isdir</code> (<i>path</i>)	Return true if the path refers to an existing directory.
<code>makedirs</code> (<i>path</i> [, <i>mode</i> , <i>exist_ok</i>])	Create a directory on the remote side.
<code>prepend_envs</code> (<i>cmd</i> [, <i>env</i>])	
<code>pull_file</code> (<i>remote_source</i> , <i>local_dir</i>)	Transport file on the remote side to a local directory
<code>push_file</code> (<i>local_source</i> , <i>remote_dir</i>)	Transport a local file to a directory on a remote machine

Attributes

<code>script_dir</code>	This is a property.
-------------------------	---------------------

5.4 Data management

<code>parsl.app.futures.DataFuture</code>	A datafuture points at an AppFuture.
<code>parsl.data_provider.data_manager.DataManager</code>	The DataManager is responsible for transferring input and output data.
<code>parsl.data_provider.staging.Staging</code>	This class defines the interface for file staging providers.
<code>parsl.data_provider.files.File</code>	The Parsl File Class.
<code>parsl.data_provider.ftp.FTPSeparateTaskStaging</code>	Performs FTP staging as a separate parsl level task.
<code>parsl.data_provider.ftp.FTPInTaskStaging</code>	Performs FTP staging as a wrapper around the application task.
<code>parsl.data_provider.file_noop.NoOpFileStaging</code>	
<code>parsl.data_provider.globus.GlobusStaging</code>	Specification for accessing data on a remote executor via Globus.
<code>parsl.data_provider.http.HTTPSeparateTaskStaging</code>	A staging provider that Performs HTTP and HTTPS staging as a separate parsl-level task.
<code>parsl.data_provider.http.HTTPInTaskStaging</code>	A staging provider that performs HTTP and HTTPS staging as in a wrapper around each task.
<code>parsl.data_provider.rsync.RSyncStaging</code>	This staging provider will execute rsync on worker nodes to stage in files from a remote location.

5.4.1 `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

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

Attributes

<code>filename</code>	Filepath of the File object this datafuture represents.
<code>filepath</code>	Filepath of the File object this datafuture represents.
<code>tid</code>	Returns the task_id of the task that will resolve this DataFuture.

5.4.2 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

<code>__init__(dfk)</code>	Initialize the DataManager.
<code>optionally_stage_in(input, func, executor)</code>	
<code>replace_task(file, func, executor)</code>	This will give staging providers the chance to wrap (or replace entirely!) the task function.
<code>replace_task_stage_out(file, func, executor)</code>	This will give staging providers the chance to wrap (or replace entirely!) the task function.

continues on next page

Table 23 – continued from previous page

<code>stage_in(file, input, executor)</code>	Transport the input from the input source to the executor, if it is file-like, returning a <code>DataFuture</code> that wraps the stage-in operation.
<code>stage_out(file, executor, app_fu)</code>	Transport the file from the local filesystem to the remote Globus endpoint.

5.4.3 `parsl.data_provider.staging.Staging`

class `parsl.data_provider.staging.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

<code>__init__()</code>	Initialize self.
<code>can_stage_in(file)</code>	Given a <code>File</code> object, decide if this staging provider can stage the file.
<code>can_stage_out(file)</code>	Like <code>can_stage_in</code> , but for staging out.
<code>replace_task(dm, executor, file, func)</code>	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.
<code>replace_task_stage_out(dm, executor, file, func)</code>	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.
<code>stage_in(dm, executor, file, parent_fut)</code>	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.
<code>stage_out(dm, executor, file, app_fu)</code>	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.

5.4.4 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.
cleancopy()	Returns a copy of the file containing only the global immutable state, without any mutable site-local local_path information.

Attributes

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

5.4.5 parsl.data_provider.ftp.FTPSeparateTaskStaging

class parsl.data_provider.ftp.**FTPSeparateTaskStaging**

Performs FTP staging as a separate parsl level task.

__init__()

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

Methods

__init__ ()	Initialize self.
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.

continues on next page

Table 27 – continued from previous page

<code>stage_in(dm, executor, file, parent_fut)</code>	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.
<code>stage_out(dm, executor, file, app_fu)</code>	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.

5.4.6 `parsl.data_provider.ftp.FTPInTaskStaging`

class `parsl.data_provider.ftp.FTPInTaskStaging`

Performs FTP staging as a wrapper around the application task.

`__init__()`

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

Methods

<code>__init__()</code>	Initialize self.
<code>can_stage_in(file)</code>	Given a File object, decide if this staging provider can stage the file.
<code>can_stage_out(file)</code>	Like <code>can_stage_in</code> , but for staging out.
<code>replace_task(dm, executor, file, f)</code>	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.
<code>replace_task_stage_out(dm, executor, file, func)</code>	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.
<code>stage_in(dm, executor, file, parent_fut)</code>	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.
<code>stage_out(dm, executor, file, app_fu)</code>	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.

5.4.7 `parsl.data_provider.file_noop.NoOpFileStaging`

class `parsl.data_provider.file_noop.NoOpFileStaging`

`__init__()`

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

Methods

<code>__init__()</code>	Initialize self.
<code>can_stage_in(file)</code>	Given a File object, decide if this staging provider can stage the file.
<code>can_stage_out(file)</code>	Like <code>can_stage_in</code> , but for staging out.
<code>replace_task(dm, executor, file, func)</code>	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.
<code>replace_task_stage_out(dm, executor, file, func)</code>	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.
<code>stage_in(dm, executor, file, parent_fut)</code>	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.
<code>stage_out(dm, executor, file, app_fu)</code>	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.

5.4.8 `parsl.data_provider.globus.GlobusStaging`

class `parsl.data_provider.globus.GlobusStaging` (*endpoint_uuid: str, endpoint_path: Optional[str] = None, local_path: Optional[str] = None*)

Specification for accessing data on a remote executor via Globus.

Parameters

- **endpoint_uuid** (*str*) – Universally unique identifier of the Globus endpoint at which the data can be accessed. This can be found in the [Manage Endpoints](#) page.
- **endpoint_path** (*str, optional*) – FIXME
- **local_path** (*str, optional*) – FIXME

__init__ (*endpoint_uuid: str, endpoint_path: Optional[str] = None, local_path: Optional[str] = None*)
Initialize self. See `help(type(self))` for accurate signature.

Methods

<code>__init__(endpoint_uuid[, endpoint_path, ...])</code>	Initialize self.
<code>can_stage_in(file)</code>	Given a File object, decide if this staging provider can stage the file.
<code>can_stage_out(file)</code>	Like <code>can_stage_in</code> , but for staging out.
<code>initialize_globus()</code>	
<code>replace_task(dm, executor, file, func)</code>	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.
<code>replace_task_stage_out(dm, executor, file, func)</code>	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.

continues on next page

Table 30 – continued from previous page

<code>stage_in(dm, executor, file, parent_fut)</code>	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.
<code>stage_out(dm, executor, file, app_fu)</code>	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.

5.4.9 `parsl.data_provider.http.HTTPSeparateTaskStaging`

class `parsl.data_provider.http.HTTPSeparateTaskStaging`

A staging provider that Performs HTTP and HTTPS staging as a separate parsl-level task. This requires a shared file system on the executor.

`__init__()`

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

Methods

<code>__init__()</code>	Initialize self.
<code>can_stage_in(file)</code>	Given a File object, decide if this staging provider can stage the file.
<code>can_stage_out(file)</code>	Like <code>can_stage_in</code> , but for staging out.
<code>replace_task(dm, executor, file, func)</code>	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.
<code>replace_task_stage_out(dm, executor, file, func)</code>	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.
<code>stage_in(dm, executor, file, parent_fut)</code>	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.
<code>stage_out(dm, executor, file, app_fu)</code>	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.

5.4.10 `parsl.data_provider.http.HTTPInTaskStaging`

class `parsl.data_provider.http.HTTPInTaskStaging`

A staging provider that performs HTTP and HTTPS staging as in a wrapper around each task. In contrast to `HTTPSeparateTaskStaging`, this provider does not require a shared file system.

`__init__()`

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

Methods

<code>__init__()</code>	Initialize self.
<code>can_stage_in(file)</code>	Given a File object, decide if this staging provider can stage the file.
<code>can_stage_out(file)</code>	Like <code>can_stage_in</code> , but for staging out.
<code>replace_task(dm, executor, file, f)</code>	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.
<code>replace_task_stage_out(dm, executor, file, func)</code>	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.
<code>stage_in(dm, executor, file, parent_fut)</code>	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.
<code>stage_out(dm, executor, file, app_fu)</code>	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.

5.4.11 `parsl.data_provider.rsync.RSyncStaging`

class `parsl.data_provider.rsync.RSyncStaging` (*hostname*)

This staging provider will execute rsync on worker nodes to stage in files from a remote location.

Worker nodes must be able to authenticate to the rsync server without interactive authentication - for example, worker initialization could include an appropriate SSH key configuration.

The submit side will need to run an rsync-compatible server (for example, an ssh server with the rsync binary installed)

`__init__` (*hostname*)

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

Methods

<code>__init__(hostname)</code>	Initialize self.
<code>can_stage_in(file)</code>	Given a File object, decide if this staging provider can stage the file.
<code>can_stage_out(file)</code>	Like <code>can_stage_in</code> , but for staging out.
<code>replace_task(dm, executor, file, f)</code>	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.
<code>replace_task_stage_out(dm, executor, file, f)</code>	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.
<code>stage_in(dm, executor, file, parent_fut)</code>	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.
<code>stage_out(dm, executor, file, parent_fut)</code>	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.

5.5 Executors

<code>parsl.executors.base.ParslExecutor</code>	Executors are abstractions that represent available compute resources to which you could submit arbitrary App tasks.
<code>parsl.executors.ThreadPoolExecutor</code>	A thread-based executor.
<code>parsl.executors.HighThroughputExecutor</code>	Executor designed for cluster-scale
<code>parsl.executors.WorkQueueExecutor</code>	Executor to use Work Queue batch system
<code>parsl.executors.ExtremeScaleExecutor</code>	Executor designed for leadership class supercomputer scale
<code>parsl.executors.LowLatencyExecutor</code>	TODO: docstring for LowLatencyExecutor
<code>parsl.executors.swift_t.TurbineExecutor</code>	The Turbine executor.

5.5.1 `parsl.executors.base.ParslExecutor`

class `parsl.executors.base.ParslExecutor`

Executors are abstractions that represent available compute resources to which you could submit arbitrary App tasks.

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 `@type-guard` cannot be persuaded otherwise. So if you're implementing an executor and want to `@type-guard` the constructor, you'll have to use `List[Any]` here.

`__init__()`

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

Methods

<code>__init__()</code>	Initialize self.
<code>create_monitoring_info(status)</code>	Create a monitoring message for each block based on the poll status.
<code>handle_errors(error_handler, status)</code>	This method is called by the error management infrastructure after a status poll.
<code>monitor_resources()</code>	Should resource monitoring happen for tasks on running on this executor?
<code>scale_in(blocks)</code>	Scale in method.

continues on next page

Table 35 – continued from previous page

<code>scale_out(blocks)</code>	Scale out method.
<code>set_bad_state_and_fail_all(exception)</code>	Allows external error handlers to mark this executor as irrecoverably bad and cause all tasks submitted to it now and in the future to fail.
<code>shutdown()</code>	Shutdown the executor.
<code>start()</code>	Start the executor.
<code>status()</code>	Return the status of all jobs/blocks currently known to this executor.
<code>submit(func, resource_specification, *args, ...)</code>	Submit.

Attributes

<code>bad_state_is_set</code>	Returns true if this executor is in an irrecoverable error state.
<code>error_management_enabled</code>	Indicates whether worker error management is supported by this executor.
<code>executor_exception</code>	Returns an exception that indicates why this executor is in an irrecoverable state.
<code>hub_address</code>	Address to the Hub for monitoring.
<code>hub_port</code>	Port to the Hub for monitoring.
<code>run_dir</code>	Path to the run directory.
<code>scaling_enabled</code>	Specify if scaling is enabled.
<code>status_polling_interval</code>	Returns the interval, in seconds, at which the status method should be called.
<code>tasks</code>	Contains a dictionary mapping task IDs to the corresponding Future objects for all tasks that have been submitted to this executor.

5.5.2 `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

<code>__init__([label, max_threads, ...])</code>	Initialize self.
<code>create_monitoring_info(status)</code>	Create a monitoring message for each block based on the poll status.
<code>handle_errors(error_handler, status)</code>	This method is called by the error management infrastructure after a status poll.
<code>monitor_resources()</code>	Resource monitoring sometimes deadlocks when using threads, so this function returns false to disable it.
<code>scale_in(blocks)</code>	Scale in the number of active blocks by specified amount.
<code>scale_out([workers])</code>	Scales out the number of active workers by 1.
<code>set_bad_state_and_fail_all(exception)</code>	Allows external error handlers to mark this executor as irrecoverably bad and cause all tasks submitted to it now and in the future to fail.
<code>shutdown([block])</code>	Shutdown the ThreadPool.
<code>start()</code>	Start the executor.
<code>status()</code>	Return the status of all jobs/blocks currently known to this executor.
<code>submit(func, resource_specification, *args, ...)</code>	Submits work to the thread pool.

Attributes

<code>bad_state_is_set</code>	Returns true if this executor is in an irrecoverable error state.
<code>error_management_enabled</code>	Indicates whether worker error management is supported by this executor.
<code>executor_exception</code>	Returns an exception that indicates why this executor is in an irrecoverable state.
<code>hub_address</code>	Address to the Hub for monitoring.
<code>hub_port</code>	Port to the Hub for monitoring.
<code>provider</code>	
<code>run_dir</code>	Path to the run directory.
<code>scaling_enabled</code>	Specify if scaling is enabled.
<code>status_polling_interval</code>	Returns the interval, in seconds, at which the status method should be called.
<code>tasks</code>	Contains a dictionary mapping task IDs to the corresponding Future objects for all tasks that have been submitted to this executor.

5.5.3 parsl.executors.HighThroughputExecutor

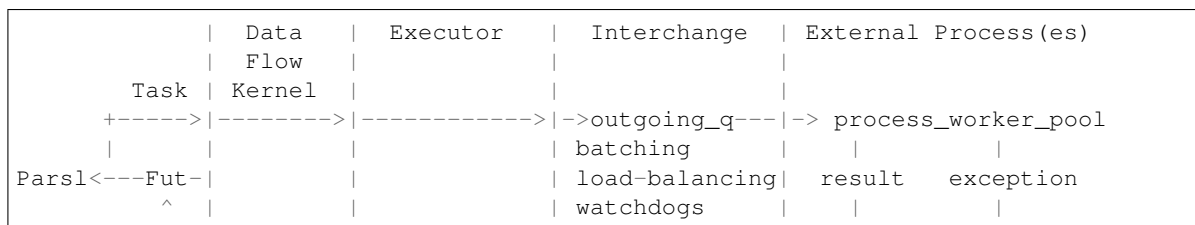
```
class parsl.executors.HighThroughputExecutor (label: str = 'HighThroughputExecutor',
                                             provider: ExecutionProvider = LocalProvider(channel=LocalChannel(envs={},
script_dir=None, user_home='/home/docs/checkouts/readthedocs.org/user_builds/parsl/che
cmd_timeout=30, init_blocks=1,
launcher=SingleNodeLauncher(debug=True,
fail_on_any=False), max_blocks=1,
min_blocks=0, move_files=None,
nodes_per_block=1, parallelism=1,
walltime='00:15:00', worker_init=""),
launch_cmd: Optional[str] = None,
address: Optional[str] = None,
worker_ports: Optional[Tuple[int,
int]] = None, worker_port_range:
Optional[Tuple[int, int]] = (54000,
55000), interchange_port_range: Op
tional[Tuple[int, int]] = (55000,
56000), storage_access: Op
tional[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, cpu_affinity: str = 'none',
prefetch_capacity: int = 0, heart
beat_threshold: int = 120, heart
beat_period: int = 30, poll_period:
int = 10, address_probe_timeout: Op
tional[int] = None, managed: bool =
True, worker_logdir_root: Optional[str] =
None)
```

Executor designed for cluster-scale

The HighThroughputExecutor system has the following components:

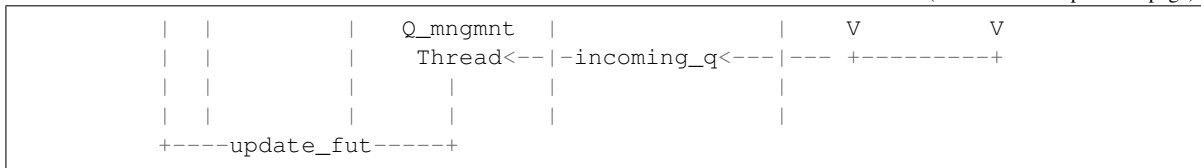
1. The HighThroughputExecutor instance which is run as part of the Parsl script.
2. The Interchange which acts as a load-balancing proxy between workers and Parsl
3. The multiprocessing based worker pool which coordinates task execution over several cores on a node.
4. ZeroMQ pipes connect the HighThroughputExecutor,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

Here is a diagram



(continues on next page)

(continued from previous page)



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`, `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. By default, the executor will attempt to enumerate and connect through all possible addresses. Setting an address here overrides the default behavior. `default=None`
- **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 there's sufficient memory for each worker. `Default: None`
- **max_workers** (*int*) – Caps the number of workers launched by the manager. `Default: infinity`

- **cpu_affinity** (*string*) – Whether or how each worker process sets thread affinity. Options are “none” to forgo any CPU affinity configuration, “block” to assign adjacent cores to workers (ex: assign 0-1 to worker 0, 2-3 to worker 1), and “alternating” to assign cores to workers in round-robin (ex: assign 0,2 to worker 0, 1,3 to worker 1).
- **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.
- **address_probe_timeout** (*int* | *None*) – Managers attempt connecting over many different addresses to determine a viable address. This option sets a time limit in seconds on the connection attempt. Default of None implies 30s timeout set on worker.
- **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, user_home='/home/docs/checkouts/readthedocs.org/user_builds/parsl/checkouts/1.1.0/docs'),
        cmd_timeout=30, init_blocks=1, launcher=SingleNodeLauncher(debug=True, fail_on_any=False), max_blocks=1, min_blocks=0, move_files=None, nodes_per_block=1,
        parallelism=1, walltime='00:15:00', worker_init=''), launch_cmd: Optional[str] = None, address: Optional[str] = None, 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, cpu_affinity: str = 'none',
        prefetch_capacity: int = 0, heartbeat_threshold: int = 120, heartbeat_period: int = 30, poll_period: int = 10, address_probe_timeout: Optional[int] = None, managed: bool =
        True, worker_logdir_root: Optional[str] = None)
```

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

Methods

<code>__init__</code> ([label, provider, launch_cmd, ...])	Initialize self.
<code>create_monitoring_info</code> (status)	Create a msg for monitoring based on the poll status
<code>handle_errors</code> (error_handler, status)	This method is called by the error management infrastructure after a status poll.
<code>hold_worker</code> (worker_id)	Puts a worker on hold, preventing scheduling of additional tasks to it.
<code>initialize_scaling</code> ()	Compose the launch command and call the scale_out
<code>monitor_resources</code> ()	Should resource monitoring happen for tasks on running on this executor?

continues on next page

Table 39 – continued from previous page

<code>scale_in([blocks, block_ids, force, ...])</code>	Scale in the number of active blocks by specified amount.
<code>scale_out([blocks])</code>	Scales out the number of blocks by “blocks”
<code>set_bad_state_and_fail_all(exception)</code>	Allows external error handlers to mark this executor as irrecoverably bad and cause all tasks submitted to it now and in the future to fail.
<code>shutdown([hub, targets, block])</code>	Shutdown the executor, including all workers and controllers.
<code>start()</code>	Create the Interchange process and connect to it.
<code>status()</code>	Return status of all blocks.
<code>submit(func, resource_specification, *args, ...)</code>	Submits work to the the outgoing_q.
<code>weakref_cb([q])</code>	We do not use this yet.

Attributes

<code>bad_state_is_set</code>	Returns true if this executor is in an irrecoverable error state.
<code>connected_managers</code>	
<code>connected_workers</code>	
<code>error_management_enabled</code>	Indicates whether worker error management is supported by this executor.
<code>executor_exception</code>	Returns an exception that indicates why this executor is in an irrecoverable state.
<code>hub_address</code>	Address to the Hub for monitoring.
<code>hub_port</code>	Port to the Hub for monitoring.
<code>outstanding</code>	
<code>provider</code>	
<code>run_dir</code>	Path to the run directory.
<code>scaling_enabled</code>	Specify if scaling is enabled.
<code>status_polling_interval</code>	Returns the interval, in seconds, at which the status method should be called.
<code>tasks</code>	Contains a dictionary mapping task IDs to the corresponding Future objects for all tasks that have been submitted to this executor.

5.5.4 parsl.executors.WorkQueueExecutor

```
class parsl.executors.WorkQueueExecutor (label: str = 'WorkQueueExecutor', provider:
parsl.providers.provider_base.ExecutionProvider
= LocalProvider(channel=LocalChannel(envs={},
script_dir=None, user-
home='/home/docs/checkouts/readthedocs.org/user_builds/parsl/checkouts/
cmd_timeout=30, init_blocks=1,
launcher=SingleNodeLauncher(debug=True,
fail_on_any=False), max_blocks=1,
min_blocks=0, move_files=None,
nodes_per_block=1, parallelism=1, wall-
time='00:15:00', worker_init="), working_dir:
str = '.', managed: bool = True, project_name:
Optional[str] = None, project_password_file:
Optional[str] = None, address: Optional[str] =
None, port: int = 9123, env: Optional[Dict] = None,
shared_fs: bool = False, storage_access: Op-
tional[List[parsl.data_provider.staging.Staging]]
= None, use_cache: bool = False, source: bool
= False, pack: bool = False, extra_pkgs: Op-
tional[List[str]] = None, autolabel: bool = False,
autolabel_window: int = 1, autocategory: bool =
True, init_command: str = "", worker_options: str
= "", full_debug: bool = True)
```

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.

Parameters

- **label** (*str*) – A human readable label for the executor, unique with respect to other Work Queue master programs. Default is “WorkQueueExecutor”.
- **working_dir** (*str*) – Location for Parsl to perform app delegation to the Work Queue system. Defaults to current directory.
- **managed** (*bool*) – Whether this executor is managed by the DFK or externally handled. Default is True (managed by DFK).
- **project_name** (*str*) – If given, Work Queue master process name. Default is None. Overrides address.
- **project_password_file** (*str*) – Optional password file for the work queue project. Default is None.
- **address** (*str*) – The ip to contact this work queue master process. If not given, uses the address of the current machine as returned by socket.gethostname(). Ignored if project_name is specified.
- **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. Default is 9123.
- **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. Default is False.
- **use_cache** (*bool*) – Whether workers should cache files that are common to tasks. Warning: Two files are considered the same if they have the same filepath name. Use with care when reusing the executor instance across multiple parsl workflows. Default is False.
- **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.) Default is False. Set to True if pack is True
- **pack** (*bool*) – Use conda-pack to prepare a self-contained Python environment for each task. This greatly increases task latency, but does not require a common environment or shared FS on execution nodes. Implies source=True.
- **extra_pkgs** (*list*) – List of extra pip/conda package names to include when packing the environment. This may be useful if the app executes other (possibly non-Python) programs provided via pip or conda. Scanning the app source for imports would not detect these dependencies, so they need to be manually specified.
- **autolabel** (*bool*) – Use the Resource Monitor to automatically determine resource labels based on observed task behavior.
- **autolabel_window** (*int*) – Set the number of tasks considered for autolabeling. Work Queue will wait for a series of N tasks with steady resource requirements before making a decision on labels. Increasing this parameter will reduce the number of failed tasks due to resource exhaustion when autolabeling, at the cost of increased resources spent collecting stats.
- **autocategory** (*bool*) – Place each app in its own category by default. If all invocations of an app have similar performance characteristics, this will provide a reasonable set of categories automatically.
- **init_command** (*str*) – Command line to run before executing a task in a worker. Default is ‘’.
- **worker_options** (*str*) – Extra options passed to work_queue_worker. Default is ‘’.

```
__init__(label: str = 'WorkQueueExecutor', provider: parsl.providers.provider_base.ExecutionProvider
        = LocalProvider(channel=LocalChannel(envs={}, script_dir=None, user-
        home='/home/docs/checkouts/readthedocs.org/user_builds/parsl/checkouts/1.1.0/docs'),
        cmd_timeout=30, init_blocks=1, launcher=SingleNodeLauncher(debug=True,
        fail_on_any=False), max_blocks=1, min_blocks=0, move_files=None, nodes_per_block=1,
        parallelism=1, walltime='00:15:00', worker_init=''), working_dir: str = '.', managed: bool
        = True, project_name: Optional[str] = None, project_password_file: Optional[str] = None,
        address: Optional[str] = None, port: int = 0, env: Optional[Dict] = None, shared_fs:
        bool = False, storage_access: Optional[List[parsl.data_provider.staging.Staging]] = None,
        use_cache: bool = False, source: bool = False, pack: bool = False, extra_pkgs: Op-
        tional[List[str]] = None, autolabel: bool = False, autolabel_window: int = 1, autocategory:
        bool = True, init_command: str = '', worker_options: str = '', full_debug: bool = True)
Initialize self. See help(type(self)) for accurate signature.
```

Methods

<code>__init__([label, provider, working_dir, ...])</code>	Initialize self.
<code>create_monitoring_info(status)</code>	Create a monitoring message for each block based on the poll status.
<code>handle_errors(error_handler, status)</code>	This method is called by the error management infrastructure after a status poll.
<code>initialize_scaling()</code>	Compose the launch command and call scale out
<code>monitor_resources()</code>	Should resource monitoring happen for tasks on running on this executor?
<code>run_dir([value])</code>	Path to the run directory.
<code>scale_in(count)</code>	Scale in method.
<code>scale_out([blocks])</code>	Scale out method.
<code>scaling_enabled()</code>	Specify if scaling is enabled.
<code>set_bad_state_and_fail_all(exception)</code>	Allows external error handlers to mark this executor as irrecoverably bad and cause all tasks submitted to it now and in the future to fail.
<code>shutdown(*args, **kwargs)</code>	Shutdown the executor.
<code>start()</code>	Create submit process and collector thread to create, send, and retrieve Parsl tasks within the Work Queue system.
<code>status()</code>	Return the status of all jobs/blocks currently known to this executor.
<code>submit(func, resource_specification, *args, ...)</code>	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

<code>bad_state_is_set</code>	Returns true if this executor is in an irrecoverable error state.
<code>error_management_enabled</code>	Indicates whether worker error management is supported by this executor.
<code>executor_exception</code>	Returns an exception that indicates why this executor is in an irrecoverable state.
<code>hub_address</code>	Address to the Hub for monitoring.
<code>hub_port</code>	Port to the Hub for monitoring.
<code>provider</code>	
<code>status_polling_interval</code>	Returns the interval, in seconds, at which the status method should be called.
<code>tasks</code>	Contains a dictionary mapping task IDs to the corresponding Future objects for all tasks that have been submitted to this executor.

5.5.5 parsl.executors.ExtremeScaleExecutor

```
class parsl.executors.ExtremeScaleExecutor (label='ExtremeScaleExecutor',
                                           provider=LocalProvider(channel=LocalChannel(envs={},
                                           script_dir=None,                               user-
                                           home='/home/docs/checkouts/readthedocs.org/user_builds/parsl/checkouts/parsl',
                                           cmd_timeout=30,                               init_blocks=1,
                                           launcher=SingleNodeLauncher(debug=True,
                                           fail_on_any=False),                               max_blocks=1,
                                           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

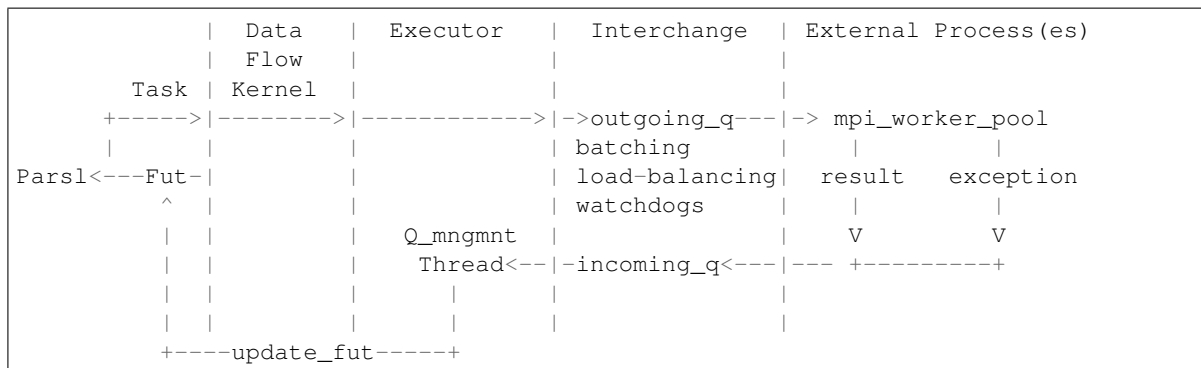
The ExtremeScaleExecutor extends the Executor interface to enable task execution on supercomputing systems (>1K Nodes). When functions and their arguments are submitted to the interface, a future is returned that tracks the execution of the function on a distributed compute environment.

The ExtremeScaleExecutor system has the following components:

1. The ExtremeScaleExecutor instance which is run as part of the Parsl script
2. The Interchange which 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



Parameters

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

Provider to access computation resources. Can be any providers in `parsl.providers`:

Cobalt, Condor, GoogleCloud, GridEngine, 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/1.1.0/docs'),
cmd_timeout=30, init_blocks=1, launcher=SingleNodeLauncher(debug=True,
fail_on_any=False), max_blocks=1, min_blocks=0, move_files=None, nodes_per_block=1,
parallelism=1, walltime='00:15:00', worker_init="), launch_cmd=None, ad-
dress='127.0.0.1', worker_ports=None, worker_port_range=(54000, 55000), in-
terchange_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

<code>__init__([label, provider, launch_cmd, ...])</code>	Initialize self.
<code>create_monitoring_info(status)</code>	Create a msg for monitoring based on the poll status
<code>handle_errors(error_handler, status)</code>	This method is called by the error management infrastructure after a status poll.
<code>hold_worker(worker_id)</code>	Puts a worker on hold, preventing scheduling of additional tasks to it.
<code>initialize_scaling()</code>	Compose the launch command and call the <code>scale_out</code>
<code>monitor_resources()</code>	Should resource monitoring happen for tasks on running on this executor?
<code>scale_in([blocks, block_ids, force, ...])</code>	Scale in the number of active blocks by specified amount.
<code>scale_out([blocks])</code>	Scales out the number of blocks by “blocks”
<code>set_bad_state_and_fail_all(exception)</code>	Allows external error handlers to mark this executor as irrecoverably bad and cause all tasks submitted to it now and in the future to fail.
<code>shutdown([hub, targets, block])</code>	Shutdown the executor, including all workers and controllers.
<code>start()</code>	Create the Interchange process and connect to it.
<code>status()</code>	Return status of all blocks.
<code>submit(func, resource_specification, *args, ...)</code>	Submits work to the the outgoing_q.
<code>weakref_cb([q])</code>	We do not use this yet.

Attributes

<code>bad_state_is_set</code>	Returns true if this executor is in an irrecoverable error state.
<code>connected_managers</code>	
<code>connected_workers</code>	
<code>error_management_enabled</code>	Indicates whether worker error management is supported by this executor.
<code>executor_exception</code>	Returns an exception that indicates why this executor is in an irrecoverable state.
<code>hub_address</code>	Address to the Hub for monitoring.
<code>hub_port</code>	Port to the Hub for monitoring.
<code>outstanding</code>	
<code>provider</code>	
<code>run_dir</code>	Path to the run directory.
<code>scaling_enabled</code>	Specify if scaling is enabled.
<code>status_polling_interval</code>	Returns the interval, in seconds, at which the status method should be called.
<code>tasks</code>	Contains a dictionary mapping task IDs to the corresponding Future objects for all tasks that have been submitted to this executor.

5.5.6 parsl.executors.LowLatencyExecutor

```
class parsl.executors.LowLatencyExecutor(label='LowLatencyExecutor',
                                         provider=LocalProvider(channel=LocalChannel(envs={},
                                         script_dir=None,
                                         userhome='/home/docs/checkouts/readthedocs.org/user_builds/parsl/checkouts/1.1.0/docs',
                                         cmd_timeout=30,
                                         init_blocks=1,
                                         launcher=SingleNodeLauncher(debug=True,
                                         fail_on_any=False),
                                         max_blocks=1,
                                         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_port=None,
                                         worker_port_range=(54000,
                                         55000),
                                         interchange_port_range=(55000,
                                         56000),
                                         working_dir=None,
                                         worker_debug=False,
                                         workers_per_node=1,
                                         managed=True)
```

TODO: docstring for LowLatencyExecutor

```
__init__(label='LowLatencyExecutor', provider=LocalProvider(channel=LocalChannel(envs={},
script_dir=None, userhome='/home/docs/checkouts/readthedocs.org/user_builds/parsl/checkouts/1.1.0/docs',
cmd_timeout=30, init_blocks=1, launcher=SingleNodeLauncher(debug=True,
fail_on_any=False), max_blocks=1, min_blocks=0, move_files=None, nodes_per_block=1,
parallelism=1, walltime='00:15:00', worker_init=''), launch_cmd=None, ad-
dress='127.0.0.1', worker_port=None, worker_port_range=(54000, 55000), inter-
change_port_range=(55000, 56000), working_dir=None, worker_debug=False, work-
ers_per_node=1, managed=True)
```

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

Methods

<code>__init__([label, provider, launch_cmd, ...])</code>	Initialize self.
<code>create_monitoring_info(status)</code>	Create a monitoring message for each block based on the poll status.
<code>handle_errors(error_handler, status)</code>	This method is called by the error management infrastructure after a status poll.
<code>monitor_resources()</code>	Should resource monitoring happen for tasks on running on this executor?
<code>scale_in(blocks)</code>	Scale in the number of active blocks by specified amount.
<code>scale_out([blocks])</code>	Scales out the number of active workers by the number of blocks specified.
<code>set_bad_state_and_fail_all(exception)</code>	Allows external error handlers to mark this executor as irrecoverably bad and cause all tasks submitted to it now and in the future to fail.
<code>shutdown([hub, targets, block])</code>	Shutdown the executor, including all workers and controllers.
<code>start()</code>	Create the Interchange process and connect to it.
<code>status()</code>	Return status of all blocks.
<code>submit(func, resource_specification, *args, ...)</code>	TODO: docstring

Attributes

<code>bad_state_is_set</code>	Returns true if this executor is in an irrecoverable error state.
<code>error_management_enabled</code>	Indicates whether worker error management is supported by this executor.
<code>executor_exception</code>	Returns an exception that indicates why this executor is in an irrecoverable state.
<code>hub_address</code>	Address to the Hub for monitoring.
<code>hub_port</code>	Port to the Hub for monitoring.
<code>provider</code>	
<code>run_dir</code>	Path to the run directory.
<code>scaling_enabled</code>	Specify if scaling is enabled.
<code>status_polling_interval</code>	Returns the interval, in seconds, at which the status method should be called.
<code>tasks</code>	Contains a dictionary mapping task IDs to the corresponding Future objects for all tasks that have been submitted to this executor.

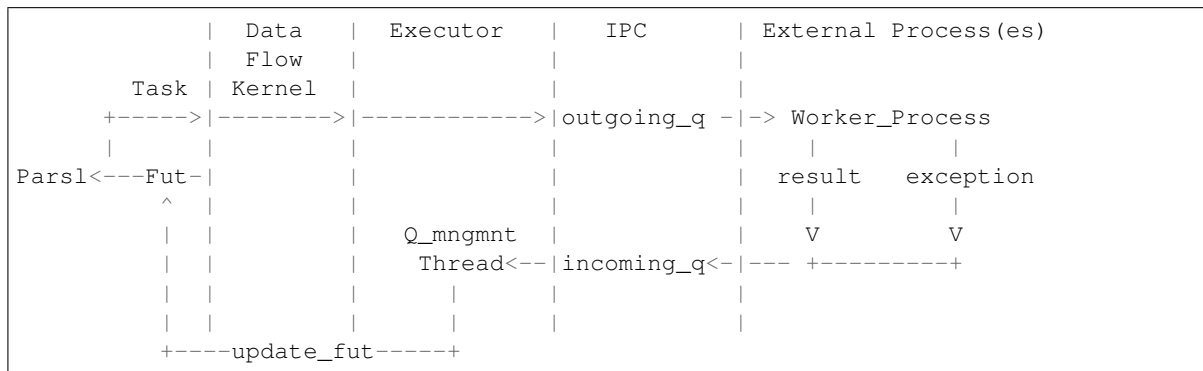
5.5.7 `parsl.executors.swift_t.TurbineExecutor`

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



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

Initialize the thread pool.

Trying to implement the emews model.

Methods

<code>__init__([label, storage_access, ...])</code>	Initialize the thread pool.
<code>create_monitoring_info(status)</code>	Create a monitoring message for each block based on the poll status.
<code>handle_errors(error_handler, status)</code>	This method is called by the error management infrastructure after a status poll.
<code>monitor_resources()</code>	Should resource monitoring happen for tasks on running on this executor?
<code>scale_in(blocks)</code>	Scale in the number of active blocks by specified amount.
<code>scale_out([blocks])</code>	Scales out the number of active workers by 1.
<code>set_bad_state_and_fail_all(exception)</code>	Allows external error handlers to mark this executor as irrecoverably bad and cause all tasks submitted to it now and in the future to fail.
<code>shutdown()</code>	Shutdown method, to kill the threads and workers.
<code>start()</code>	Start the executor.
<code>status()</code>	Return the status of all jobs/blocks currently known to this executor.
<code>submit(func, *args, **kwargs)</code>	Submits work to the the outgoing_q.
<code>weakref_cb([q])</code>	We do not use this yet.

Attributes

<code>bad_state_is_set</code>	Returns true if this executor is in an irrecoverable error state.
<code>error_management_enabled</code>	Indicates whether worker error management is supported by this executor.
<code>executor_exception</code>	Returns an exception that indicates why this executor is in an irrecoverable state.
<code>hub_address</code>	Address to the Hub for monitoring.
<code>hub_port</code>	Port to the Hub for monitoring.
<code>provider</code>	
<code>run_dir</code>	Path to the run directory.
<code>scaling_enabled</code>	Specify if scaling is enabled.
<code>status_polling_interval</code>	Returns the interval, in seconds, at which the status method should be called.
<code>tasks</code>	Contains a dictionary mapping task IDs to the corresponding Future objects for all tasks that have been submitted to this executor.

5.6 Launchers

<code>parsl.launchers.SimpleLauncher</code>	Does no wrapping.
<code>parsl.launchers.SingleNodeLauncher</code>	Worker launcher that wraps the user's command with the framework to launch multiple command invocations in parallel.
<code>parsl.launchers.SrunLauncher</code>	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.
<code>parsl.launchers.AprunLauncher</code>	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
<code>parsl.launchers.SrunMPILauncher</code>	Launches as many workers as MPI tasks to be executed concurrently within a block.
<code>parsl.launchers.GnuParallelLauncher</code>	Worker launcher that wraps the user's command with the framework to launch multiple command invocations via GNU parallel sshlogin.
<code>parsl.launchers.MpiExecLauncher</code>	Worker launcher that wraps the user's command with the framework to launch multiple command invocations via mpiexec.
<code>parsl.launchers.JsrunLauncher</code>	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
<code>parsl.launchers.WrappedLauncher</code>	Wraps the command by prepending commands before a user's command

5.6.1 parsl.launchers.SimpleLauncher

class `parsl.launchers.SimpleLauncher` (*debug: bool = True*)

Does no wrapping. Just returns the command as-is

__init__ (*debug: bool = True*)

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

Methods

<code>__init__</code> ([debug])	Initialize self.
---------------------------------	------------------

5.6.2 parsl.launchers.SingleNodeLauncher

class `parsl.launchers.SingleNodeLauncher` (*debug: bool = True, fail_on_any: bool = False*)

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__ (*debug: bool = True, fail_on_any: bool = False*)

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

Methods

<code>__init__</code> ([debug, fail_on_any])	Initialize self.
--	------------------

5.6.3 `parsl.launchers.SrunLauncher`

class `parsl.launchers.SrunLauncher` (*debug*: *bool* = *True*, *overrides*: *str* = "")

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__` (*debug*: *bool* = *True*, *overrides*: *str* = "")

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

Methods

<code>__init__</code> ([debug, overrides])	param overrides This string will be passed to the srun launcher. Default: “
--	--

5.6.4 `parsl.launchers.AprunLauncher`

class `parsl.launchers.AprunLauncher` (*debug*: *bool* = *True*, *overrides*: *str* = "")

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__` (*debug*: *bool* = *True*, *overrides*: *str* = "")

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

Methods

<code>__init__</code> ([debug, overrides])	param overrides This string will be passed to the aprun launcher. Default: “
--	---

5.6.5 `parsl.launchers.SrunMPILauncher`

class `parsl.launchers.SrunMPILauncher` (*debug: bool = True, overrides: str = ''*)

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__` (*debug: bool = True, overrides: str = ''*)

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

Methods

`__init__` ([*debug*, *overrides*])

param `overrides` This string will be passed to the launcher. Default: ''

5.6.6 `parsl.launchers.GnuParallelLauncher`

class `parsl.launchers.GnuParallelLauncher` (*debug: bool = True*)

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 `$PATH`
- Passwordless SSH login is configured between the controller node and the target nodes.
- The provider makes available the `$PBS_NODEFILE` environment variable

`__init__` (*debug: bool = True*)

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

Methods

`__init__` ([*debug*])

Initialize self.

5.6.7 `parsl.launchers.MpiExecLauncher`

class `parsl.launchers.MpiExecLauncher` (*debug: bool = True*)

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__(debug: bool = True)
```

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

Methods

<code>__init__([debug])</code>	Initialize self.
--------------------------------	------------------

5.6.8 parsl.launchers.JsrunLauncher

class `parsl.launchers.JsrunLauncher` (*debug: bool = True, overrides: str = ''*)
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__(debug: bool = True, overrides: str = '')
```

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

Methods

<code>__init__([debug, overrides])</code>	param overrides This string will be passed to the JSrun launcher. Default: ''
---	--

5.6.9 parsl.launchers.WrappedLauncher

class `parsl.launchers.WrappedLauncher` (*prepend: str, debug: bool = True*)
Wraps the command by prepending commands before a user's command

As an example, the wrapped launcher can be used to launch a command inside a docker container by prepending the proper docker invocation

```
__init__(prepend: str, debug: bool = True)
```

Parameters `prepend` (*str*) – Command to use before the launcher (e.g., time)

Methods

<code>__init__(prepend[, debug])</code>	param prepend Command to use before the launcher (e.g., time)
---	--

5.7 Providers

<code>parsl.providers.AdHocProvider</code>	Ad-hoc execution provider
<code>parsl.providers.AWSProvider</code>	A provider for using Amazon Elastic Compute Cloud (EC2) resources.
<code>parsl.providers.CobaltProvider</code>	Cobalt Execution Provider
<code>parsl.providers.CondorProvider</code>	HTCondor Execution Provider.
<code>parsl.providers.GoogleCloudProvider</code>	A provider for using resources from the Google Compute Engine.
<code>parsl.providers.GridEngineProvider</code>	A provider for the Grid Engine scheduler.
<code>parsl.providers.LocalProvider</code>	Local Execution Provider
<code>parsl.providers.LSFProvider</code>	LSF Execution Provider
<code>parsl.providers.GridEngineProvider</code>	A provider for the Grid Engine scheduler.
<code>parsl.providers.SlurmProvider</code>	Slurm Execution Provider
<code>parsl.providers.TorqueProvider</code>	Torque Execution Provider
<code>parsl.providers.KubernetesProvider</code>	Kubernetes execution provider
<code>parsl.providers.PBSProProvider</code>	PBS Pro Execution Provider
<code>parsl.providers.provider_base.ExecutionProvider</code>	Execution providers are responsible for managing execution resources that have a Local Resource Manager (LRM).
<code>parsl.providers.cluster_provider.ClusterProvider</code>	This class defines behavior common to all cluster/supercompute-style scheduler systems.

5.7.1 `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 objects*) – 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

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

Attributes

<code>cores_per_node</code>	Number of cores to provision per node.
<code>label</code>	Provides the label for this provider
<code>mem_per_node</code>	Real memory to provision per node in GB.
<code>scaling_enabled</code>	
<code>status_polling_interval</code>	Returns the interval, in seconds, at which the status method should be called.

5.7.2 parsl.providers.AWSProvider

```
class parsl.providers.AWSProvider(image_id, key_name, init_blocks=1, min_blocks=0,
                                max_blocks=10, nodes_per_block=1, paral-
                                lism=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(debug=True,
                                fail_on_any=False))
```

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 (optionally) 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(debug=True, fail_on_any=False))
Initialize self. See help(type(self)) for accurate signature.
```

Methods

<code>__init__(image_id, key_name[, init_blocks, ...])</code>	Initialize self.
<code>cancel(job_ids)</code>	Cancel the jobs specified by a list of job ids.
<code>config_route_table(vpc, internet_gateway)</code>	Configure route table for Virtual Private Cloud (VPC).
<code>create_name_tag_spec(resource_type, name)</code>	Create a new tag specification for a resource name.
<code>create_session()</code>	Create a session.
<code>create_vpc()</code>	Create and configure VPC
<code>generate_aws_id()</code>	Generate a new ID for AWS resources.
<code>get_instance_state([instances])</code>	Get states of all instances on EC2 which were started by this file.
<code>goodbye()</code>	
<code>initialize_boto_client()</code>	Initialize the boto client.
<code>read_state_file(state_file)</code>	Read the state file, if it exists.
<code>security_group(vpc, name)</code>	Create and configure a new security group.
<code>show_summary()</code>	Print human readable summary of current AWS state to log and to console.
<code>shut_down_instance([instances])</code>	Shut down a list of instances, if provided.
<code>spin_up_instance(command, job_name)</code>	Start an instance in the VPC in the first available subnet.
<code>status(job_ids)</code>	Get the status of a list of jobs identified by their ids.

continues on next page

Table 62 – continued from previous page

<code>submit([command, tasks_per_node, job_name])</code>	Submit the command onto a freshly instantiated AWS EC2 instance.
<code>teardown()</code>	Teardown the EC2 infrastructure.
<code>write_state_file()</code>	Save information that must persist to a file.
<code>xstr(s)</code>	

Attributes

<code>cores_per_node</code>	Number of cores to provision per node.
<code>label</code>	Provides the label for this provider
<code>mem_per_node</code>	Real memory to provision per node in GB.
<code>status_polling_interval</code>	Returns the interval, in seconds, at which the status method should be called.

5.7.3 parsl.providers.CobaltProvider

```
class parsl.providers.CobaltProvider(channel=LocalChannel(envs={},
script_dir=None, user=
home='/home/docs/checkouts/readthedocs.org/user_builds/parsl/checkouts/1.1.0',
nodes_per_block=1, init_blocks=0, min_blocks=0,
max_blocks=1, parallelism=1, walltime='00:10:00',
account=None, queue=None, scheduler_options="",
worker_init="", launcher=AprunLauncher(debug=True,
overrides=""), cmd_timeout=10)
```

Cobalt Execution Provider

This provider uses cobalt to submit (qsub), obtain the status of (qstat), and cancel (qdel) jobs. The 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/1.1.0/docs'),
        nodes_per_block=1, init_blocks=0, min_blocks=0, max_blocks=1, parallelism=1, wall-
        time='00:10:00', account=None, queue=None, scheduler_options="", worker_init="",
        launcher=AprunLauncher(debug=True, overrides=""), cmd_timeout=10)
Initialize self. See help(type(self)) for accurate signature.
```

Methods

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

Attributes

<code>cores_per_node</code>	Number of cores to provision per node.
<code>label</code>	Provides the label for this provider
<code>mem_per_node</code>	Real memory to provision per node in GB.
<code>status_polling_interval</code>	Returns the interval, in seconds, at which the status method should be called.

5.7.4 parsl.providers.CondorProvider

```
class parsl.providers.CondorProvider(channel: parsl.channels.base.Channel = LocalChannel(envs={}, script_dir=None, user-
        home='/home/docs/checkouts/readthedocs.org/user_builds/parsl/checkouts/1.1.0/docs'),
        nodes_per_block: int = 1, cores_per_slot: Optional[float] = None, mem_per_slot: Optional[float] = None,
        init_blocks: int = 1, min_blocks: int = 0, max_blocks: int = 1, 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(debug=True, fail_on_any=False),
        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 environment 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/1.1.0/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 = 1,
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(debug=True, fail_on_any=False), requirements: str = "", cmd_timeout: int = 60)
→ None
```

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

Methods

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

Attributes

<code>cores_per_node</code>	Number of cores to provision per node.
<code>label</code>	Provides the label for this provider
<code>mem_per_node</code>	Real memory to provision per node in GB.
<code>status_polling_interval</code>	Returns the interval, in seconds, at which the status method should be called.

5.7.5 `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(debug=True,  
                                         fail_on_any=False), 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. Possibilities 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.

```
__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(debug=True, fail_on_any=False), parallelism=1)  
Initialize self. See help(type(self)) for accurate signature.
```

Methods

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

Attributes

<code>status_polling_interval</code>

5.7.6 parsl.providers.GridEngineProvider

```
class parsl.providers.GridEngineProvider(channel=LocalChannel(envs={},
    script_dir=None,                                user_home='/home/docs/checkouts/readthedocs.org/user_builds/parsl/checkout',
    nodes_per_block=1,                               init_blocks=1,
    min_blocks=0,                                     max_blocks=1,       parallelism=1,
    walltime='00:10:00',
    scheduler_options="",                             worker_init="",
    launcher=SingleNodeLauncher(debug=True,
    fail_on_any=False), cmd_timeout: int = 60,
    queue=None)
```

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),
- **cmd_timeout** (*int*) – Timeout for commands made to the scheduler in seconds

```
__init__ (channel=LocalChannel(envs={}, script_dir=None, user-
home='/home/docs/checkouts/readthedocs.org/user_builds/parsl/checkouts/1.1.0/docs'),
nodes_per_block=1, init_blocks=1, min_blocks=0, max_blocks=1, par-
allelism=1, walltime='00:10:00', scheduler_options="", worker_init="",
launcher=SingleNodeLauncher(debug=True, fail_on_any=False), cmd_timeout: int =
60, queue=None)
```

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

Methods

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

<code>cores_per_node</code>	Number of cores to provision per node.
<code>label</code>	Provides the label for this provider
<code>mem_per_node</code>	Real memory to provision per node in GB.
<code>status_polling_interval</code>	Returns the interval, in seconds, at which the status method should be called.

5.7.7 parsl.providers.LocalProvider

```
class parsl.providers.LocalProvider (channel=LocalChannel(envs={}, script_dir=None, user-
home='/home/docs/checkouts/readthedocs.org/user_builds/parsl/checkouts/1.1.0/
nodes_per_block=1, launcher=SingleNodeLauncher(debug=True,
fail_on_any=False), init_blocks=1, min_blocks=0,
max_blocks=1, 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/1.1.0/docs'),
        nodes_per_block=1, launcher=SingleNodeLauncher(debug=True, fail_on_any=False),
        init_blocks=1, min_blocks=0, max_blocks=1, walltime='00:15:00', worker_init="",
        cmd_timeout=30, parallelism=1, move_files=None)
Initialize self. See help(type(self)) for accurate signature.
```

Methods

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

Attributes

<code>cores_per_node</code>	Number of cores to provision per node.
<code>label</code>	Provides the label for this provider
<code>mem_per_node</code>	Real memory to provision per node in GB.
<code>status_polling_interval</code>	Returns the interval, in seconds, at which the status method should be called.

5.7.8 parsl.providers.LSFProvider

```
class parsl.providers.LSFProvider(channel=LocalChannel(envs={}, script_dir=None, user-
        home='/home/docs/checkouts/readthedocs.org/user_builds/parsl/checkouts/1.1.0/doc
        nodes_per_block=1, init_blocks=1, min_blocks=0,
        max_blocks=1, parallelism=1, wall-
        time='00:10:00', scheduler_options="", worker_init="",
        project=None, cmd_timeout=120, move_files=True,
        launcher=SingleNodeLauncher(debug=True,
        fail_on_any=False))
```

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, user_home='/home/docs/checkouts/readthedocs.org/user_builds/parsl/checkouts/1.1.0/docs'),
         nodes_per_block=1, init_blocks=1, min_blocks=0, max_blocks=1, parallelism=1, walltime='00:10:00', scheduler_options="", worker_init="", project=None, cmd_timeout=120,
         move_files=True, launcher=SingleNodeLauncher(debug=True, fail_on_any=False))
Initialize self. See help(type(self)) for accurate signature.
```

Methods

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

Attributes

<code>cores_per_node</code>	Number of cores to provision per node.
<code>label</code>	Provides the label for this provider
<code>mem_per_node</code>	Real memory to provision per node in GB.
<code>status_polling_interval</code>	Returns the interval, in seconds, at which the status method should be called.

5.7.9 parsl.providers.SlurmProvider

```
class parsl.providers.SlurmProvider (partition: Optional[str], account: Optional[str]
                                   = None, channel: parsl.channels.base.Channel
                                   = LocalChannel(envs={}, script_dir=None, user-
                                   home='/home/docs/checkouts/readthedocs.org/user_builds/parsl/checkouts/1.1.0/
                                   nodes_per_block: int = 1, cores_per_node: Op-
                                   tional[int] = None, mem_per_node: Optional[int]
                                   = None, init_blocks: int = 1, min_blocks: int = 0,
                                   max_blocks: int = 1, parallelism: float = 1, wall-
                                   time: str = '00:10:00', scheduler_options: str = "",
                                   worker_init: str = "", cmd_timeout: int = 10, exclu-
                                   sive: bool = True, move_files: bool = True, launcher:
                                   parsl.launchers.launchers.Launcher = SingleNode-
                                   Launcher(debug=True, fail_on_any=False))
```

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. If none, no partition slurm directive will be specified.
- **account** (*str*) – Slurm account to which to charge resources used by the job. If none, the job will use the user's default account.
- **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** (*int*) – 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: Optional[str], account: Optional[str] = None, channel:
parsl.channels.base.Channel = LocalChannel(envs={}, script_dir=None, user-
home='/home/docs/checkouts/readthedocs.org/user_builds/parsl/checkouts/1.1.0/docs'),
nodes_per_block: int = 1, cores_per_node: Optional[int] = None, mem_per_node:
Optional[int] = None, init_blocks: int = 1, min_blocks: int = 0, max_blocks: int
= 1, parallelism: float = 1, walltime: str = '00:10:00', scheduler_options: str = "",
worker_init: str = "", cmd_timeout: int = 10, exclusive: bool = True, move_files: bool =
True, launcher: parsl.launchers.launchers.Launcher = SingleNodeLauncher(debug=True,
fail_on_any=False))
```

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

Methods

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

Attributes

<code>cores_per_node</code>	Number of cores to provision per node.
<code>label</code>	Provides the label for this provider
<code>mem_per_node</code>	Real memory to provision per node in GB.
<code>status_polling_interval</code>	Returns the interval, in seconds, at which the status method should be called.

5.7.10 parsl.providers.TorqueProvider

```
class parsl.providers.TorqueProvider(channel=LocalChannel(envs={}),
                                     script_dir=None,                               user-
                                     home='/home/docs/checkouts/readthedocs.org/user_builds/parsl/checkouts/1.1.0/docs',
                                     account=None, queue=None, scheduler_options="",
                                     worker_init="", nodes_per_block=1, init_blocks=1,
                                     min_blocks=0, max_blocks=1, parallelism=1,
                                     launcher=AprunLauncher(debug=True, 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. WARNING: scheduler_options should only be given #PBS strings, and should not have trailing newlines.
- **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/1.1.0/docs',
         account=None, queue=None, scheduler_options="", worker_init="",
         nodes_per_block=1, init_blocks=1, min_blocks=0, max_blocks=1, parallelism=1,
         launcher=AprunLauncher(debug=True, overrides=""), walltime='00:20:00',
         cmd_timeout=120)
```

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

Methods

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

Attributes

<code>cores_per_node</code>	Number of cores to provision per node.
<code>label</code>	Provides the label for this provider
<code>mem_per_node</code>	Real memory to provision per node in GB.
<code>status_polling_interval</code>	Returns the interval, in seconds, at which the status method should be called.

5.7.11 `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, se-  
                                         cret: Optional[str] = None, persistent_volumes:  
                                         List[Tuple[str, str]] = [])
```

Kubernetes execution provider

Parameters

- **namespace** (*str*) – Kubernetes namespace to create deployments.
- **image** (*str*) – Docker image to use in the deployment.
- **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.
- **max_blocks** (*int*) – Maximum number of blocks to maintain.
- **max_cpu** (*float*) – 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.
- **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

<code>__init__(image[, namespace, ...])</code>	Initialize self.
<code>cancel(job_ids)</code>	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.
<code>status(job_ids)</code>	Get the status of a list of jobs identified by the job identifiers returned from the submit request.
<code>submit(cmd_string, tasks_per_node[, job_name])</code>	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

<code>cores_per_node</code>	Number of cores to provision per node.
<code>label</code>	Provides the label for this provider
<code>mem_per_node</code>	Real memory to provision per node in GB.
<code>status_polling_interval</code>	Returns the interval, in seconds, at which the status method should be called.

5.7.12 `parsl.providers.PBSProProvider`

```
class parsl.providers.PBSProProvider (channel=LocalChannel(envs={},
                                script_dir=None,                                user-
                                home='/home/docs/checkouts/readthedocs.org/user_builds/parsl/checkouts/1.1.0',
                                account=None, queue=None, scheduler_options="",
                                worker_init="", nodes_per_block=1, cpus_per_node=1,
                                init_blocks=1, min_blocks=0, max_blocks=1, paral-
                                lelism=1, launcher=SingleNodeLauncher(debug=True,
                                fail_on_any=False),                               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, user-
        home='/home/docs/checkouts/readthedocs.org/user_builds/parsl/checkouts/1.1.0/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=1, parallelism=1,
        launcher=SingleNodeLauncher(debug=True, fail_on_any=False), walltime='00:20:00',
        cmd_timeout=120)
    Initialize self. See help(type(self)) for accurate signature.
```

Methods

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

Attributes

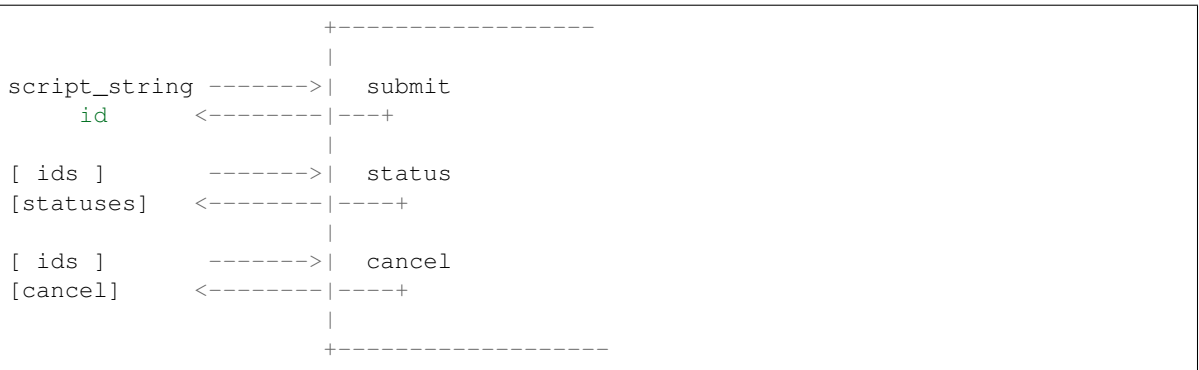
<code>cores_per_node</code>	Number of cores to provision per node.
<code>label</code>	Provides the label for this provider
<code>mem_per_node</code>	Real memory to provision per node in GB.
<code>status_polling_interval</code>	Returns the interval, in seconds, at which the status method should be called.

5.7.13 parsl.providers.provider_base.ExecutionProvider

class `parsl.providers.provider_base.ExecutionProvider`

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.

The providers abstract away the interfaces provided by various systems to request, monitor, and cancel compute resources.



```
__init__()
```

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

Methods

<code>__init__()</code>	Initialize self.
<code>cancel(job_ids)</code>	Cancels the resources identified by the <code>job_ids</code> provided by the user.
<code>status(job_ids)</code>	Get the status of a list of jobs identified by the job identifiers returned from the submit request.
<code>submit(command, tasks_per_node[, job_name])</code>	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

<code>cores_per_node</code>	Number of cores to provision per node.
<code>label</code>	Provides the label for this provider
<code>mem_per_node</code>	Real memory to provision per node in GB.
<code>status_polling_interval</code>	Returns the interval, in seconds, at which the status method should be called.

5.7.14 `parsl.providers.cluster_provider.ClusterProvider`

```
class parsl.providers.cluster_provider.ClusterProvider(label, channel,
                                                    nodes_per_block,
                                                    init_blocks, min_blocks,
                                                    max_blocks, parallelism, walltime, launcher,
                                                    cmd_timeout=10)
```

This class defines behavior common to all cluster/supercompute-style scheduler systems.

Parameters

- **label** (*str*) – Label for this provider.
- **channel** (*Channel*) – Channel for accessing this provider. Possible channels include *LocalChannel* (the default), *SSHChannel*, or *SSHInteractiveLoginChannel*.
- **walltime** (*str*) – Walltime requested per block in HH:MM:SS.
- **launcher** (*str*) – FIXME
- **cmd_timeout** (*int*) – Timeout for commands made to the scheduler in seconds

```

+-----+
|
script_string ----->| submit
  id      <-----|---+
|
[ ids ]      ----->| status
[ statuses] <-----|----+
|

```

(continues on next page)

(continued from previous page)

```

[ ids ]      ----->|  cancel
[cancel]     <-----|-----+
                |
                +-----+

```

__init__ (*label, channel, nodes_per_block, init_blocks, min_blocks, max_blocks, parallelism, wall-time, launcher, cmd_timeout=10*)
Initialize self. See help(type(self)) for accurate signature.

Methods

<code>__init__(label, channel, nodes_per_block, ...)</code>	Initialize self.
<code>cancel(job_ids)</code>	Cancels the resources identified by the job_ids provided by the user.
<code>execute_wait(cmd[, timeout])</code>	
<code>status(job_ids)</code>	Get the status of a list of jobs identified by the job identifiers returned from the submit request.
<code>submit(command, tasks_per_node[, job_name])</code>	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

<code>cores_per_node</code>	Number of cores to provision per node.
<code>label</code>	Provides the label for this provider
<code>mem_per_node</code>	Real memory to provision per node in GB.
<code>status_polling_interval</code>	Returns the interval, in seconds, at which the status method should be called.

5.8 Exceptions

<code>parsl.app.errors.AppBadFormatting</code>	An error raised during formatting of a bash function.
<code>parsl.app.errors.AppException</code>	An error raised during execution of an app.
<code>parsl.app.errors.AppTimeout</code>	An error raised during execution of an app when it exceeds its allotted walltime.
<code>parsl.app.errors.BadStdStreamFile</code>	Error raised due to bad filepaths specified for STDOUT/STDERR.
<code>parsl.app.errors.BashAppNoReturn</code>	Bash app returned no string.
<code>parsl.app.errors.BashExitFailure</code>	A non-zero exit code returned from a @bash_app
<code>parsl.app.errors.MissingOutputs</code>	Error raised at the end of app execution due to missing output files.
<code>parsl.app.errors.NotFutureError</code>	A non future item was passed to a function that expected a future.
<code>parsl.app.errors.ParslError</code>	Base class for all exceptions.

continues on next page

Table 88 – continued from previous page

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

5.8.1 `parsl.app.errors.AppBadFormatting`

exception `parsl.app.errors.AppBadFormatting`
 An error raised during formatting of a bash function.

5.8.2 `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

5.8.3 `parsl.app.errors.AppTimeout`

exception `parsl.app.errors.AppTimeout`
 An error raised during execution of an app when it exceeds its allotted walltime.

5.8.4 `parsl.app.errors.BadStdStreamFile`

exception `parsl.app.errors.BadStdStreamFile` (*reason: str, exception: Exception*)
 Error raised due to bad filepaths specified for STDOUT/ STDERR.
Contains: reason(string) outputs(List of strings/files..) exception object

5.8.5 `parsl.app.errors.BashAppNoReturn`

exception `parsl.app.errors.BashAppNoReturn` (*reason: str*)
 Bash app returned no string.
 Contains: reason(string)

5.8.6 `parsl.app.errors.BashExitFailure`

exception `parsl.app.errors.BashExitFailure` (*reason: str, exitcode: int*)
 A non-zero exit code returned from a @bash_app
 Contains: reason(str) exitcode(int)

5.8.7 `parsl.app.errors.MissingOutputs`

exception `parsl.app.errors.MissingOutputs` (*reason: str, outputs: List[Union[str, parsl.data_provider.files.File]]*)
 Error raised at the end of app execution due to missing output files.
 Contains: reason(string) outputs(List of strings/files..)

5.8.8 `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.

5.8.9 `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.

5.8.10 `parsl.errors.OptionalModuleMissing`

exception `parsl.errors.OptionalModuleMissing` (*module_names*, *reason*)
Error raised when a required module is missing for a optional/extra component

5.8.11 `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.

5.8.12 `parsl.executors.errors.ScalingFailed`

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

5.8.13 `parsl.executors.errors.SerializationError`

exception `parsl.executors.errors.SerializationError` (*fname*)
Failure to serialize data arguments for the tasks

5.8.14 `parsl.executors.errors.DeserializationError`

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

5.8.15 `parsl.executors.errors.BadMessage`

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

5.8.16 `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.

5.8.17 `parsl.dataflow.error.ConfigurationError`

exception `parsl.dataflow.error.ConfigurationError`
Raised when the DataFlowKernel receives an invalid configuration.

5.8.18 `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.

5.8.19 `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*

5.8.20 `parsl.dataflow.error.DependencyError`

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

Error raised if an app cannot run because there was an error in a dependency.

Parameters

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

Contains: *reason* (string) *dependent_exceptions*

5.8.21 `parsl.launchers.error.BadLauncher`

exception `parsl.launchers.error.BadLauncher` (*launcher, reason*)
Error raised when a non callable object is provider as Launcher

5.8.22 `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.

5.8.23 `parsl.providers.error.ChannelRequired`

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

5.8.24 `parsl.providers.error.ScaleOutFailed`

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

5.8.25 `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

5.8.26 `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

5.8.27 `parsl.channels.errors.ChannelError`

exception `parsl.channels.errors.ChannelError` (*reason: str, e: Exception, hostname: str*)
Base class for all exceptions
Only to be invoked when only a more specific error is not available.

5.8.28 `parsl.channels.errors.BadHostKeyException`

exception `parsl.channels.errors.BadHostKeyException` (*e: Exception, hostname: str*)
 SSH channel could not be created since server's host keys could not be verified

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

5.8.29 `parsl.channels.errors.BadScriptPath`

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

5.8.30 `parsl.channels.errors.BadPermsScriptPath`

exception `parsl.channels.errors.BadPermsScriptPath` (*e: Exception, hostname: str*)
 User does not have permissions to access the script_dir on the remote site

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

5.8.31 `parsl.channels.errors.FileExists`

exception `parsl.channels.errors.FileExists` (*e: Exception, hostname: str, filename: Optional[str] = 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)

5.8.32 `parsl.channels.errors.AuthException`

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

5.8.33 `parsl.channels.errors.SSHException`

exception `parsl.channels.errors.SSHException` (*e: Exception, hostname: str*)
 if there was any other error connecting or establishing an SSH session

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

5.8.34 `parsl.channels.errors.FileCopyException`

exception `parsl.channels.errors.FileCopyException` (*e: Exception, hostname: str*)
 File copy operation failed

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

5.8.35 `parsl.executors.high_throughput.errors.WorkerLost`

exception `parsl.executors.high_throughput.errors.WorkerLost` (*worker_id*, *host-name*)
Exception raised when a worker is lost

5.9 Internal

<code>parsl.app.app.AppBase</code>	This is the base class that defines the two external facing functions that an App must define.
<code>parsl.app.bash.BashApp</code>	
<code>parsl.app.python.PythonApp</code>	Extends <code>AppBase</code> to cover the Python App.
<code>parsl.dataflow.dflow.DataFlowKernel</code>	The <code>DataFlowKernel</code> adds dependency awareness to an existing executor.
<code>parsl.dataflow.flow_control.FlowControl</code>	Implements threshold-interval based flow control.
<code>parsl.dataflow.memoization.Memoizer</code>	Memoizer is responsible for ensuring that identical work is not repeated.
<code>parsl.dataflow.strategy.Strategy</code>	FlowControl strategy.
<code>parsl.dataflow.flow_control.Timer</code>	This timer is a simplified version of the FlowControl timer.

5.9.1 `parsl.app.app.AppBase`

class `parsl.app.app.AppBase` (*func*, *data_flow_kernel=None*, *executors='all'*, *cache=False*, *ignore_for_cache=None*)

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.

__init__ (*func*, *data_flow_kernel=None*, *executors='all'*, *cache=False*, *ignore_for_cache=None*)
Construct the App object.

Parameters **func** (-) – Takes the function to be made into an App

Kwargs:

- **data_flow_kernel** (`DataFlowKernel`): The `DataFlowKernel` responsible for managing this app. This can be omitted only after calling `parsl.dataflow.dflow.DataFlowKernelLoader.load()`.
- **executors** (strlist) : Labels of the executors that this app can execute over. Default is 'all'.
- **cache** (Bool) : Enable caching of this app ?

Returns

- App object.

Methods

<code>__init__(func[, data_flow_kernel, ...])</code>	Construct the App object.
--	---------------------------

5.9.2 parsl.app.bash.BashApp

class `parsl.app.bash.BashApp` (*func*, *data_flow_kernel=None*, *cache=False*, *executors='all'*, *ignore_for_cache=None*)

__init__ (*func*, *data_flow_kernel=None*, *cache=False*, *executors='all'*, *ignore_for_cache=None*)
Construct the App object.

Parameters **func** (-) – Takes the function to be made into an App

Kwargs:

- *data_flow_kernel* (*DataFlowKernel*): The *DataFlowKernel* responsible for managing this app. This can be omitted only after calling `parsl.dataflow.dflow.DataFlowKernelLoader.load()`.
- *executors* (strlist) : Labels of the executors that this app can execute over. Default is 'all'.
- *cache* (Bool) : Enable caching of this app ?

Returns

- App object.

Methods

<code>__init__(func[, data_flow_kernel, cache, ...])</code>	Construct the App object.
---	---------------------------

5.9.3 parsl.app.python.PythonApp

class `parsl.app.python.PythonApp` (*func*, *data_flow_kernel=None*, *cache=False*, *executors='all'*, *ignore_for_cache=[], join=False*)

Extends AppBase to cover the Python App.

__init__ (*func*, *data_flow_kernel=None*, *cache=False*, *executors='all'*, *ignore_for_cache=[], join=False*)
Construct the App object.

Parameters **func** (-) – Takes the function to be made into an App

Kwargs:

- *data_flow_kernel* (*DataFlowKernel*): The *DataFlowKernel* responsible for managing this app. This can be omitted only after calling `parsl.dataflow.dflow.DataFlowKernelLoader.load()`.
- *executors* (strlist) : Labels of the executors that this app can execute over. Default is 'all'.
- *cache* (Bool) : Enable caching of this app ?

Returns

- App object.

Methods

<code>__init__(func[, data_flow_kernel, cache, ...])</code>	Construct the App object.
---	---------------------------

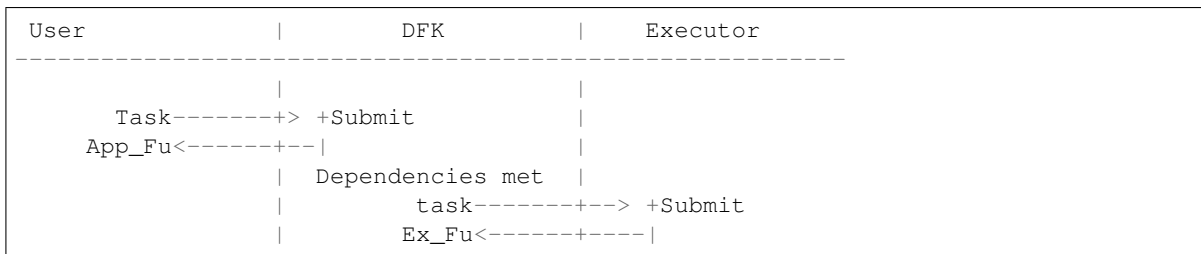
5.9.4 `parsl.dataflow.dflow.DataFlowKernel`

```
class parsl.dataflow.dflow.DataFlowKernel (config=Config(app_cache=True,      check-
                                          point_files=None,    checkpoint_mode=None,
                                          checkpoint_period=None,    execu-
                                          tors=[ThreadPoolExecutor(label='threads',
                                                                  managed=True,    max_threads=2,    stor-
                                                                  age_access=None,    thread_name_prefix="",
                                                                  working_dir=None)],    garbage_collect=True,
                                          initialize_logging=True,    in-
                                          ternal_tasks_max_threads=10,
                                          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:



```
__init__ (config=Config(app_cache=True, checkpoint_files=None, checkpoint_mode=None, check-
point_period=None, executors=[ThreadPoolExecutor(label='threads', managed=True,
max_threads=2, storage_access=None, thread_name_prefix="", working_dir=None)],
garbage_collect=True, initialize_logging=True, internal_tasks_max_threads=10,
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.

Methods

<code>__init__([config])</code>	Initialize the DataFlowKernel.
<code>add_executors(executors)</code>	
<code>atexit_cleanup()</code>	
<code>check_staging_inhibited(kwargs)</code>	
<code>checkpoint([tasks])</code>	Checkpoint the dfk incrementally to a checkpoint file.
<code>cleanup()</code>	DataFlowKernel cleanup.
<code>handle_app_update(task_id, future)</code>	This function is called as a callback when an AppFuture is in its final state.
<code>handle_exec_update(task_id, future)</code>	This function is called only as a callback from an execution attempt reaching a final state (either successfully or failing).
<code>handle_join_update(outer_task_id, ...)</code>	
<code>launch_if_ready(task_id)</code>	<code>launch_if_ready</code> will launch the specified task, if it is ready to run (for example, without dependencies, and in pending state).
<code>launch_task(task_id, executable, *args, **kwargs)</code>	Handle the actual submission of the task to the executor layer.
<code>load_checkpoints(checkpointDirs)</code>	Load checkpoints from the checkpoint files into a dictionary.
<code>log_task_states()</code>	
<code>sanitize_and_wrap(task_id, args, kwargs)</code>	This function should be called only when all the futures we track have been resolved.
<code>submit(func, app_args[, executors, cache, ...])</code>	Add task to the dataflow system.
<code>wait_for_current_tasks()</code>	Waits for all tasks in the task list to be completed, by waiting for their AppFuture to be completed.
<code>wipe_task(task_id)</code>	Remove task with <code>task_id</code> from the internal tasks table

Attributes

<code>config</code>	Returns the fully initialized config that the DFK is actively using.
---------------------	--

5.9.5 `parsl.dataflow.flow_control.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) :
```

(continues on next page)

(continued from previous page)

```
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 as queue

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.

`__init__` (*dfk*, **args*, *threshold*=20, *interval*=5)

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

Methods

<code>__init__</code> (<i>dfk</i> , * <i>args</i> [, <i>threshold</i> , <i>interval</i>])	Initialize the flowcontrol object.
<code>add_executors</code> (<i>executors</i>)	
<code>close</code> ()	Merge the threads and terminate.
<code>make_callback</code> ([<i>kind</i>])	Makes the callback and resets the timer.
<code>notify</code> (<i>event_id</i>)	Let the FlowControl system know that there is an event.

5.9.6 parsl.dataflow.memoization.Memoizer

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](#)

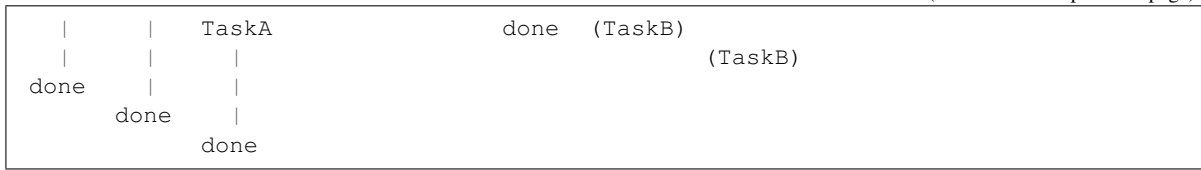
The memoizer implementation here does not collapse duplicate calls at call time, but works **only** when the result of a previous call is available at the time the duplicate call is made.

For instance:

No advantage from memoization here:	Memoization helps here:
TaskA TaskA	TaskB

(continues on next page)

(continued from previous page)



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.

Methods

<code>__init__(dfk[, memoize, checkpoint])</code>	Initialize the memoizer.
<code>check_memo(task_id, task)</code>	Create a hash of the task and its inputs and check the lookup table for this hash.
<code>hash_lookup(hashsum)</code>	Lookup a hash in the memoization table.
<code>make_hash(task)</code>	Create a hash of the task inputs.
<code>update_memo(task_id, task, r)</code>	Updates the memoization lookup table with the result from a task.

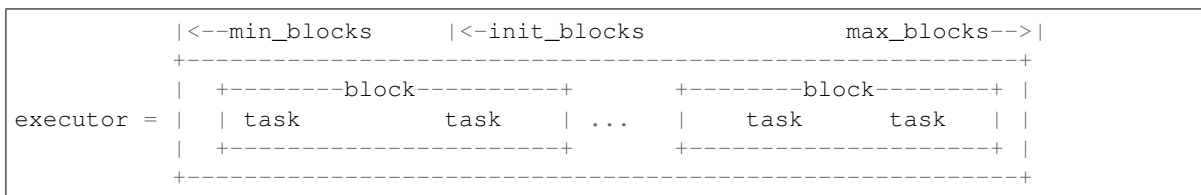
5.9.7 parsl.dataflow.strategy.Strategy

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.



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

```
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.

Methods

<code>__init__(dfk)</code>	Initialize strategy.
<code>add_executors(executors)</code>	
<code>unset_logging()</code>	Mute newly added handlers to the root level, right after calling <code>executor.status</code>

5.9.8 `parsl.dataflow.flow_control.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

Methods

<code>__init__(callback, *args[, interval, name])</code>	Initialize the flowcontrol object We start the timer thread here
<code>close()</code>	Merge the threads and terminate.
<code>make_callback([kind])</code>	Makes the callback and resets the timer.

DEVELOPER DOCUMENTATION

6.1 Contributing

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

If you're interested in contributing, please review our [contributing guide](#).

6.2 Changelog

6.2.1 Parsl 1.1.0

Released on April 26th, 2021.

Parsl v1.1.0 includes 59 closed issues and 243 pull requests with contributions (code, tests, reviews and reports) from:

Akila Ravihansa Perera @ravihansa3000, Anna Woodard @annawoodard, @bakerjl, Ben Clifford @benclifford, Daniel S. Katz @danielskatz, Douglas Thain @dthain, @gerrick, @JG-Quarknet, Joseph Moon @jmoon1506, Kelly L. Rowland @kellyrowland, Lars Bilke @bilke, Logan Ward @WardLT, Kirill Nagaitsev @Loonride, Marcus Schwarting @meschw04, Matt Baughman @mattebaughman, Mihael Hategan @hategan, @radiantone, Rohan Kumar @rohankumar42, Sohiti Miglani @sohitmiglani, Tim Shaffer @trshaffer, Tyler J. Skluzacek @tskluzac, Yadu Nand Babuji @yadudoc, and Zhuozhao Li @ZhuozhaoLi

Deprecated and Removed features

- Python 3.5 is no longer supported.
- Almost definitely broken Jetstream provider removed (#1821)

New Functionality

- Allow HTEX to set CPU affinity (#1853)
- New serialization system to replace IPP serialization (#1806)
- Support for Python 3.9
- @join_apps are a variation of @python_apps where an app can launch more apps and then complete only after the launched apps are also completed.

These are described more fully in docs/userguide/joins.rst

- **Monitoring:** `hub.log` is now named `monitoring_router.log` Remove denormalised workflow duration from monitoring db (#1774) Remove hostname from status table (#1847) Clarify distinction between tasks and tries to run tasks (#1808) Replace ‘done’ state with ‘exec_done’ and ‘memo_done’ (#1848) Use repr instead of str for monitoring fail history (#1966)
- **Monitoring visualization:** Make task list appear under `.../task/` not under `.../app/` (#1762) Test that `parsl-visualize` does not return HTTP errors (#1700) Generate Gantt chart from status table rather than task table timestamps (#1767) Hyperlinks for app page to task pages should be on the task ID, not the app name (#1776) Use real final state to color DAG visualization (#1812)
- Make task record garbage collection optional. (#1909)
- Make `checkpoint_files = get_all_checkpoints()` by default (#1918)

6.2.2 Parsl 1.0.0

Released on June 11th, 2020

Parsl v1.0.0 includes 59 closed issues and 243 pull requests with contributions (code, tests, reviews and reports) from:

Akila Ravihansa Perera @ravihansa3000, Aymen Alsaadi @AymenFJA, Anna Woodard @annawoodard, Ben Clifford @benclifford, Ben Glick @benhg, Benjamin Tovar @btovar, Daniel S. Katz @danielskatz, Daniel Smith @dga-smith, Douglas Thain @dthain, Eric Jonas @ericmjonas, Geoffrey Lentner @glentner, Ian Foster @ianfoster, Kalpani Ranasinghe @kalpanibhagya, Kyle Chard @kylechard, Lindsey Gray @lgray, Logan Ward @WardLT, Lyle Hayhurst @lhayhurst, Mihael Hategan @hategan, Rajini Wijayawardana @rajiniw95, @saktar-unr, Tim Shaffer @trshaffer, Tom Glanzman @TomGlanzman, Yadu Nand Babuji @yadudoc and, Zhuozhao Li @ZhuozhaoLi

Deprecated and Removed features

- **Python3.5** is now marked for deprecation, and will not be supported after this release. Python3.6 will be the earliest Python3 version supported in the next release.
- **App** decorator deprecated in 0.8 is now removed [issue#1539](#) `bash_app` and `python_app` are the only supported App decorators in this release.
- **IPyParallelExecutor** is no longer a supported executor [issue#1565](#)

New Functionality

- `WorkQueueExecutor` introduced in v0.9.0 is now in beta. `WorkQueueExecutor` is designed as a drop-in replacement for `HighThroughputExecutor`. Here are some key features: * Support for packaging the python environment and shipping it to the worker side. This mechanism addresses propagating python environments in grid-like systems that lack shared-file systems or cloud environments. * `WorkQueueExecutor` supports resource function tagging and resource specification * Support for resource specification kwarg [issue#1675](#)
- Limited type-checking in Parsl internal components (as part of an ongoing effort)
- Improvements to caching mechanism including ability to mark certain arguments to be not counted for memoization.
 - Normalize known types for memoization, and reject unknown types (#1291). This means that previous unreliable behaviour for some complex types such as dicts will become more reliable; and that other previous unreliable behaviour for other unknown complex types will now cause an error. Handling can be added for those types using `parsl.memoization.id_for_memo`.

- Add ability to label some arguments in an app invocation as not memoized using the `ignore_for_cache` app keyword (PR 1568)
- Special keyword args: ‘inputs’, ‘outputs’ that are used to specify files no longer support strings and now require `File` objects. For example, the following snippet is no longer supported in v1.0.0:

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

concat = cat(inputs=['hello-0.txt'],
             outputs=['hello-1.txt'])
```

This is the new syntax:

```
from parsl import File

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

concat = cat(inputs=[File('hello-0.txt')],
             outputs=[File('hello-1.txt')])
```

Since filenames are no longer passed to apps as strings, and the string filepath is required, it can be accessed from the File object using the `filepath` attribute.

```
from parsl import File

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

- New launcher: `WrappedLauncher` for launching tasks inside containers.
- `SSHChannel` now supports a `key_filename` kwarg [issue#1639](#)
- Newly added Makefile wraps several frequent developer operations such as:
 - Run the test-suite: `make test`
 - Install parsl: `make install`
 - Create a virtualenv: `make virtualenv`
 - Tag release and push to release channels: `make deploy`
- Several updates to the `HighThroughputExecutor`:
 - By default, the `HighThroughputExecutor` will now use heuristics to detect and try all addresses when the workers connect back to the parsl master. An address can be configured manually using the `HighThroughputExecutor(address=<address_string>)` kwarg option.
 - Support for Mac OS. ([pull#1469](#), [pull#1738](#))
 - Cleaner reporting of version mismatches and automatic suppression of non-critical errors.
 - Separate worker log directories by block id [issue#1508](#)
- Support for garbage collection to limit memory consumption in long-lived scripts.
- All cluster providers now use `max_blocks=1` by default [issue#1730](#) to avoid over-provisioning.
- New `JobStatus` class for better monitoring of Jobs submitted to batch schedulers.

Bug Fixes

- Ignore AUTO_LOGNAME for caching [issue#1642](#)
- Add batch jobs to PBS/torque job status table [issue#1650](#)
- Use higher default buffer threshold for serialization [issue#1654](#)
- Do not pass mutable default to ignore_for_cache [issue#1656](#)
- Several improvements and fixes to Monitoring
- Fix sites/test_ec2 failure when aws user opts specified [issue#1375](#)
- Fix LocalProvider to kill the right processes, rather than all processes owned by user [issue#1447](#)
- Exit htex probe loop with first working address [issue#1479](#)
- Allow slurm partition to be optional [issue#1501](#)
- Fix race condition with wait_for_tasks vs task completion [issue#1607](#)
- Fix Torque job_id truncation [issue#1583](#)
- Cleaner reporting for Serialization Errors [issue#1355](#)
- Results from zombie managers do not crash the system, but will be ignored [issue#1665](#)
- Guarantee monitoring will send out at least one message [issue#1446](#)
- Fix monitoring ctrlc hang [issue#1670](#)

6.2.3 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
```

(continues on next page)

(continued from previous page)

```
@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.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](#).

6.2.4 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.
- `File` no longer behaves as a string. This means that operations in apps that treated a `File` as a string 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](#)

6.2.5 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 *KubernetesProvider*.
- Scaling recommendations from study on Bluewaters is now available in the userguide.

6.2.6 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](#)

6.2.7 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 @jmwozniak, 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
```

(continues on next page)

(continued from previous page)

```

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)

- Ctrl+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](#)

6.2.8 Parsl 0.6.1

Released on July 23rd, 2018.

This point release contains fixes for [issue#409](#)

6.2.9 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>),
        )
    ]
)
```

- Implicit Data Staging [issue#281](#)

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

# 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()
```

- 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 inconsistencies 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.

6.2.10 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](#)

6.2.11 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](#)

6.2.12 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": "ap1_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](#)

6.2.13 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.

6.2.14 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](#)

6.2.15 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.

6.2.16 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](#)

6.2.17 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

6.2.18 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 [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 $\sim \text{length_of_iterable}/1000$ (items/task_launch_rate).

```
futures = {};

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

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

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

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

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-file systems 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 :

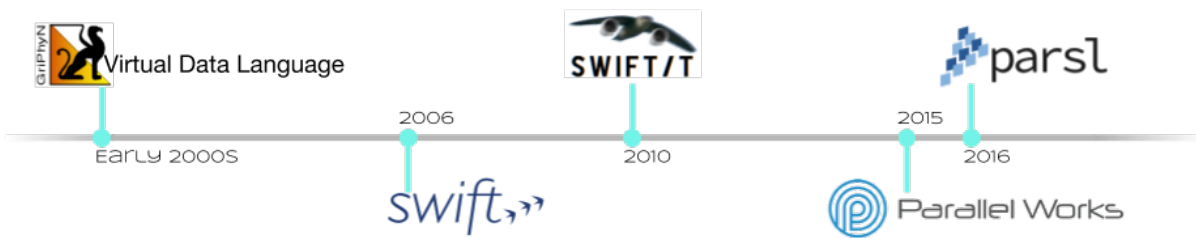
Compute Resource	Python3.4	Python3.5	Python3.6
Midway (RCC, UChicago)	X	X	
Open Science Grid	X	X	
BlueWaters	X	X	
AWS/Google Cloud	X	X	X
Beagle	X		

Design

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.

TODO: Add diagram for staging

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 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`
3. Commit and push the changes to github
4. 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
```

(continues on next page)

(continued from previous page)

```
# 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/*
```

6.7 Doc Docs

6.7.1 Documentation location

Documentation is maintained in Python docstrings throughout the code. These are imported via the [autodoc](#) Sphinx extension in `docs/reference.rst`. Individual stubs for user-facing classes (located in `stubs`) are generated automatically via `sphinx-autogen`. Parsl modules, classes, and methods can be cross-referenced from a docstring by enclosing it in backticks (```).

6.7.2 Remote builds

Builds are automatically performed by `readthedocs.io` and published to `parsl.readthedocs.io` upon git commits.

6.7.3 Local builds

To build the documentation locally, use:

```
$ make html
```

6.7.4 Regenerate module stubs

If necessary, docstring stubs can be regenerated using:

```
$ sphinx-autogen reference.rst
```


INDICES AND TABLES

- `genindex`
- `modindex`
- `search`

Symbols

`__init__()` (*parsl.app.app.AppBase* method), 164
`__init__()` (*parsl.app.bash.BashApp* method), 165
`__init__()` (*parsl.app.futures.DataFuture* method), 110
`__init__()` (*parsl.app.python.PythonApp* method), 165
`__init__()` (*parsl.channels.LocalChannel* method), 106
`__init__()` (*parsl.channels.OAuthSSHChannel* method), 108
`__init__()` (*parsl.channels.SSHChannel* method), 107
`__init__()` (*parsl.channels.SSHInteractiveLoginChannel* method), 109
`__init__()` (*parsl.channels.base.Channel* method), 105
`__init__()` (*parsl.config.Config* method), 102
`__init__()` (*parsl.data_provider.data_manager.DataManager* method), 111
`__init__()` (*parsl.data_provider.file_noop.NoOpFileStaging* method), 114
`__init__()` (*parsl.data_provider.files.File* method), 113
`__init__()` (*parsl.data_provider.ftp.FTPInTaskStaging* method), 114
`__init__()` (*parsl.data_provider.ftp.FTPSeparateTaskStaging* method), 113
`__init__()` (*parsl.data_provider.globus.GlobusStaging* method), 115
`__init__()` (*parsl.data_provider.http.HTTPInTaskStaging* method), 116
`__init__()` (*parsl.data_provider.http.HTTPSeparateTaskStaging* method), 116
`__init__()` (*parsl.data_provider.rsync.RSyncStaging* method), 117
`__init__()` (*parsl.data_provider.staging.Staging* method), 112
`__init__()` (*parsl.dataflow.dflow.DataFlowKernel* method), 166
`__init__()` (*parsl.dataflow.dflow.DataFlowKernelLoader* method), 99
`__init__()` (*parsl.dataflow.flow_control.FlowControl* method), 168
`__init__()` (*parsl.dataflow.flow_control.Timer* method), 171
`__init__()` (*parsl.dataflow.futures.AppFuture* method), 98
`__init__()` (*parsl.dataflow.memoization.Memoizer* method), 169
`__init__()` (*parsl.dataflow.strategy.Strategy* method), 170
`__init__()` (*parsl.executors.ExtremeScaleExecutor* method), 129
`__init__()` (*parsl.executors.HighThroughputExecutor* method), 123
`__init__()` (*parsl.executors.LowLatencyExecutor* method), 131
`__init__()` (*parsl.executors.ThreadPoolExecutor* method), 119
`__init__()` (*parsl.executors.WorkQueueExecutor* method), 126
`__init__()` (*parsl.executors.base.ParslExecutor* method), 118
`__init__()` (*parsl.executors.swift_t.TurbineExecutor* method), 132
`__init__()` (*parsl.launchers.AprunLauncher* method), 135
`__init__()` (*parsl.launchers.GnuParallelLauncher* method), 136
`__init__()` (*parsl.launchers.JsrunLauncher* method), 137
`__init__()` (*parsl.launchers.MpiExecLauncher* method), 136
`__init__()` (*parsl.launchers.SimpleLauncher* method), 134
`__init__()` (*parsl.launchers.SingleNodeLauncher* method), 134
`__init__()` (*parsl.launchers.SrunLauncher* method), 135
`__init__()` (*parsl.launchers.SrunMPILauncher* method), 136
`__init__()` (*parsl.launchers.WrappedLauncher* method), 137

`__init__()` (*parsl.monitoring.MonitoringHub method*), 100

`__init__()` (*parsl.providers.AWSPProvider method*), 140

`__init__()` (*parsl.providers.AdHocProvider method*), 138

`__init__()` (*parsl.providers.CobaltProvider method*), 141

`__init__()` (*parsl.providers.CondorProvider method*), 143

`__init__()` (*parsl.providers.GoogleCloudProvider method*), 144

`__init__()` (*parsl.providers.GridEngineProvider method*), 146

`__init__()` (*parsl.providers.KubernetesProvider method*), 153

`__init__()` (*parsl.providers.LSFProvider method*), 148

`__init__()` (*parsl.providers.LocalProvider method*), 147

`__init__()` (*parsl.providers.PBSPProProvider method*), 154

`__init__()` (*parsl.providers.SlurmProvider method*), 150

`__init__()` (*parsl.providers.TorqueProvider method*), 151

`__init__()` (*parsl.providers.cluster_provider.ClusterProvider method*), 157

`__init__()` (*parsl.providers.provider_base.ExecutionProvider method*), 155

A

`address_by_hostname()` (*in module parsl.addresses*), 103

`address_by_interface()` (*in module parsl.addresses*), 104

`address_by_query()` (*in module parsl.addresses*), 104

`address_by_route()` (*in module parsl.addresses*), 104

`AdHocProvider` (*class in parsl.providers*), 138

`AppBadFormatting`, 159

`AppBase` (*class in parsl.app.app*), 164

`AppException`, 159

`AppFuture` (*class in parsl.dataflow.futures*), 98

`AppTimeout`, 159

`AprunLauncher` (*class in parsl.launchers*), 135

`AuthException`, 163

`AWSPProvider` (*class in parsl.providers*), 139

B

`BadCheckpoint`, 161

`BadHostKeyException`, 163

`BadLauncher`, 162

`BadMessage`, 161

`BadPermsScriptPath`, 163

`BadScriptPath`, 163

`BadStdStreamFile`, 159

`bash_app()` (*in module parsl.app.app*), 98

`BashApp` (*class in parsl.app.bash*), 165

`BashAppNoReturn`, 159

`BashExitFailure`, 159

C

`Channel` (*class in parsl.channels.base*), 105

`ChannelError`, 162

`ChannelRequired`, 162

`ClusterProvider` (*class in parsl.providers.cluster_provider*), 156

`CobaltProvider` (*class in parsl.providers*), 141

`CondorProvider` (*class in parsl.providers*), 142

`Config` (*class in parsl.config*), 101

`ConfigurationError`, 161

D

`DataFlowException`, 161

`DataFlowKernel` (*class in parsl.dataflow.dflow*), 166

`DataFlowKernelLoader` (*class in parsl.dataflow.dflow*), 99

`DataFuture` (*class in parsl.app.futures*), 110

`DataManager` (*class in parsl.data_provider.data_manager*), 111

`DependencyError`, 161

`DeserializationError`, 160

`DuplicateTaskError`, 161

E

`ExecutionProvider` (*class in parsl.providers.provider_base*), 155

`ExecutionProviderException`, 162

`ExecutorError`, 160

`ExtremeScaleExecutor` (*class in parsl.executors*), 128

F

`File` (*class in parsl.data_provider.files*), 113

`FileCopyException`, 163

`FileExists`, 163

`FlowControl` (*class in parsl.dataflow.flow_control*), 167

`FTPInTaskStaging` (*class in parsl.data_provider.ftp*), 114

`FTPSeparateTaskStaging` (*class in parsl.data_provider.ftp*), 113

G

`get_all_checkpoints()` (*in module parsl.utils*), 104

`get_last_checkpoint()` (in module `parsl.utils`), 104

`GlobusStaging` (class in `parsl.data_provider.globus`), 115

`GnuParallelLauncher` (class in `parsl.launchers`), 136

`GoogleCloudProvider` (class in `parsl.providers`), 144

`GridEngineProvider` (class in `parsl.providers`), 145

H

`HighThroughputExecutor` (class in `parsl.executors`), 121

`HTTPInTaskStaging` (class in `parsl.data_provider.http`), 116

`HTTPSeparateTaskStaging` (class in `parsl.data_provider.http`), 116

J

`join_app()` (in module `parsl.app.app`), 98

`JsrnLauncher` (class in `parsl.launchers`), 137

K

`KubernetesProvider` (class in `parsl.providers`), 152

L

`LocalChannel` (class in `parsl.channels`), 106

`LocalProvider` (class in `parsl.providers`), 146

`LowLatencyExecutor` (class in `parsl.executors`), 131

`LSFProvider` (class in `parsl.providers`), 147

M

`Memoizer` (class in `parsl.dataflow.memoization`), 168

`MissingOutputs`, 159

`MonitoringHub` (class in `parsl.monitoring`), 100

`MpiExecLauncher` (class in `parsl.launchers`), 136

N

`NoOpFileStaging` (class in `parsl.data_provider.file_noop`), 114

`NotFutureError`, 160

O

`OAuthSSHChannel` (class in `parsl.channels`), 108

`OptionalModuleMissing`, 160

P

`ParslError`, 160

`ParslExecutor` (class in `parsl.executors.base`), 118

`PBSPROProvider` (class in `parsl.providers`), 154

`python_app()` (in module `parsl.app.app`), 97

`PythonApp` (class in `parsl.app.python`), 165

R

`RSyncStaging` (class in `parsl.data_provider.rsync`), 117

S

`ScaleOutFailed`, 162

`ScalingFailed`, 160

`SchedulerMissingArgs`, 162

`ScriptPathError`, 162

`SerializationError`, 160

`set_file_logger()` (in module `parsl`), 103

`set_stream_logger()` (in module `parsl`), 103

`SimpleLauncher` (class in `parsl.launchers`), 134

`SingleNodeLauncher` (class in `parsl.launchers`), 134

`SlurmProvider` (class in `parsl.providers`), 149

`SrunLauncher` (class in `parsl.launchers`), 135

`SrunMPILauncher` (class in `parsl.launchers`), 136

`SSHChannel` (class in `parsl.channels`), 107

`SSHException`, 163

`SSHInteractiveLoginChannel` (class in `parsl.channels`), 109

`Staging` (class in `parsl.data_provider.staging`), 112

`Strategy` (class in `parsl.dataflow.strategy`), 169

T

`ThreadPoolExecutor` (class in `parsl.executors`), 119

`Timer` (class in `parsl.dataflow.flow_control`), 171

`TorqueProvider` (class in `parsl.providers`), 151

`TurbineExecutor` (class in `parsl.executors.swift_t`), 132

W

`WorkerLost`, 164

`WorkQueueExecutor` (class in `parsl.executors`), 125

`WrappedLauncher` (class in `parsl.launchers`), 137