
Parsl Documentation

Release 1.3.0-dev

The Parsl Team

Apr 26, 2024

CONTENTS

1	Why use Parsl?	3
1.1	Parsl is Python	3
1.2	Parsl works everywhere	3
1.3	Parsl is flexible	3
1.4	Parsl handles data	3
1.5	Parsl is fast	3
1.6	Parsl is a community	4
2	Indices and tables	273
	Index	275

Parsl extends parallelism in Python beyond a single computer.

You can use Parsl just like Python's parallel executors but across *multiple cores and nodes*. However, the real power of Parsl is in expressing multi-step workflows of functions. Parsl lets you chain functions together and will launch each function as inputs and computing resources are available.

```
import parsl
from parsl import python_app

# Start Parsl on a single computer
parsl.load()

# Make functions parallel by decorating them
@python_app
def f(x):
    return x + 1

@python_app
def g(x):
    return x * 2

# These functions now return Futures, and can be chained
future = f(1)
assert future.result() == 2

future = g(f(1))
assert future.result() == 4
```

Start with the [configuration quickstart](#) to learn how to tell Parsl how to use your computing resource, see if a [template configuration for your supercomputer](#) is already available, then explore the [parallel computing patterns](#) to determine how to use parallelism best in your application.

Parsl is an open-source code, and available on GitHub: <https://github.com/parsl/parsl/>

WHY USE PARSL?

1.1 Parsl is Python

Everything about a Parsl program is written in Python. Parsl follows Python's native parallelization approach and functions, how they combine into workflows, and where they run are all described in Python.

1.2 Parsl works everywhere

Parsl can run parallel functions on a laptop and the world's fastest supercomputers. Scaling from laptop to supercomputer is often as simple as changing the resource configuration. Parsl is tested [on many of the top supercomputers](#).

1.3 Parsl is flexible

Parsl supports many kinds of applications. Functions can be pure Python or invoke external codes, be single- or multi-threaded or GPUs.

1.4 Parsl handles data

Parsl has first-class support for workflows involving files. Data will be automatically moved between workers, even if they reside on different filesystems.

1.5 Parsl is fast

Parsl was built for speed. Workflows can manage tens of thousands of parallel tasks and process thousands of tasks per second.

1.6 Parsl is a community

Parsl is part of a large, experienced community.

The Parsl Project was launched by researchers with decades of experience in workflows as part of a National Science Foundation project to create sustainable research software.

The Parsl team is guided by the community through its GitHub, conversations on [Slack](#), Bi-Weekly developer calls, and engagement with the [Workflows Community Initiative](#).

1.6.1 Table of Contents

Quickstart

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

Installation

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

Parsl requires Python3.8+ and has been tested on Linux and macOS.

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`

Installation using Conda

1. Create and activate a new conda environment:

```
$ conda create --name parsl_py38 python=3.8  
$ source activate parsl_py38
```

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.

Getting started



Parsl has much in common with Python’s native concurrency library, but unlocking Parsl’s potential requires understanding a few major concepts.

A Parsl program submits tasks to run on Workers distributed across remote computers. The instructions for these tasks are contained within “*apps*” that users define using Python functions. Each remote computer (e.g., a node on a supercomputer) has a single “*Executor*” which manages the workers. Remote resources available to Parsl are acquired by a “*Provider*”, which places the executor on a system with a “*Launcher*”. Task execution is brokered by a “*Data Flow Kernel*” that runs on your local system.

We describe these components briefly here, and link to more details in the [User Guide](#).

Note: Parsl’s documentation includes [templates for many supercomputers](#). Even though you may not need to write a configuration from a blank slate, understanding the basic terminology below will be very useful.

Application Types

Parsl enables concurrent execution of Python functions (`python_app`) or external applications (`bash_app`). The logic for both are described by Python functions marked with Parsl decorators. When decorated functions are invoked, they run asynchronously on other resources. The result of a call to a Parsl app is an `AppFuture`, which behaves like a Python Future.

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

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

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

with parsl.load():
    # 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

```

(continues on next page)

(continued from previous page)

```
hello_bash('World (Bash)').result()

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

Learn more about the types of Apps and their options [here](#).

Executors

Executors define how Parsl deploys work on a computer. Many types are available, each with different advantages.

The *HighThroughputExecutor*, like Python's `ProcessPoolExecutor`, creates workers that are separate Python processes. However, you have much more control over how the work is deployed. You can dynamically set the number of workers based on available memory and pin each worker to specific GPUs or CPU cores among other powerful features.

Learn more about Executors [here](#).

Execution Providers

Resource providers allow Parsl to gain access to computing power. For supercomputers, gaining resources often requires requesting them from a scheduler (e.g., Slurm). Parsl Providers write the requests to requisition “**Blocks**” (e.g., supercomputer nodes) on your behalf. Parsl comes pre-packaged with Providers compatible with most supercomputers and some cloud computing services.

Another key role of Providers is defining how to start an Executor on a remote computer. Often, this simply involves specifying the correct Python environment and (described below) how to launch the Executor on each acquired computers.

Learn more about Providers [here](#).

Launchers

The Launcher defines how to spread workers across all nodes available in a Block. A common example is an `MPILauncher`, which uses MPI's mechanism for starting a single program on multiple computing nodes. Like Providers, Parsl comes packaged with Launchers for most supercomputers and clouds.

Learn more about Launchers [here](#).

Benefits of a Data-Flow Kernel

The Data-Flow Kernel (DFK) is the behind-the-scenes engine behind Parsl. The DFK determines when tasks can be started and sends them to open resources, receives results, restarts failed tasks, propagates errors to dependent tasks, and performs the many other functions needed to execute complex workflows. The flexibility and performance of the DFK enables applications with intricate dependencies between tasks to execute on thousands of parallel workers.

Start with the Tutorial or the [parallel patterns](#) to see the complex types of workflows you can make with Parsl.

Starting Parsl

A Parsl script must contain the function definitions, resource configuration, and a call to `parsl.load` before launching tasks. This script runs on a system that must stay on-line until all of your tasks complete but need not have much computing power, such as the login node for a supercomputer.

The `Config` object holds definitions of Executors and the Providers and Launchers they rely on. An example which launches 512 workers on 128 nodes of the Polaris supercomputer looks like

```
config = Config(
    retries=1, # Restart task if they fail once
    executors=[
        HighThroughputExecutor(
            available_accelerators=4, # Maps one worker per GPU
            address=address_by_hostname(),
            cpu_affinity="alternating", # Prevents thread contention
            provider=PBSPROProvider(
                account="example",
                worker_init="module load conda; conda activate parsl",
                walltime="1:00:00",
                queue="prod",
                scheduler_options="#PBS -l filesystems=home:eagle", # Change if data on
↪ other filesystem
                launcher=MpiExecLauncher(
                    bind_cmd="--cpu-bind", overrides="--depth=64 --ppn 1"
                ), # Ensures 1 manger per node and allows it to divide work to all 64
↪ cores
                select_options="ngpus=4",
                nodes_per_block=128,
                cpus_per_node=64,
            ),
        ],
    )
```

The documentation has examples for other supercomputers [here](#).

The next step is to load the configuration

```
parsl.load(config)
```

You are then ready to use 10 PFLOPS of computing power through Python!

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](#).

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](#)*.

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_package1>...]'
```

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__)
```

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)
```

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`).

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())
```

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())
```

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())
```

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.

AppFutures

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

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

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

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()))
```

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
↪'))])
```

(continues on next page)

(continued from previous page)

```

# 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()))

```

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.

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())

```

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

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

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

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.

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())
```

(continues on next page)

(continued from previous page)

```

# 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())

```

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)

```

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=[]):

```

(continues on next page)

(continued from previous page)

```

    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'))])

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

```

Dynamic workflows with apps that generate other apps

Often there is a need for a workflow to launch apps based on results from prior 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.

Parsl's `join_app` is designed to address this situation by allowing you to define sub-workflows. Rather than return a value (like `python_app`) a `join_app` instead returns a future. When invoked, the `join_app` will not complete until the future has completed and the return value will be the return value from the future.

The following example shows how recursive Fibonacci can be implemented using a `join_app`. Here the `fibonacci` app makes calls to a separate `add` app for each pair of numbers.

```

[ ]: from parsl.app.app import join_app, python_app

@python_app
def add(*args):
    """Add all of the arguments together. If no arguments, then
    zero is returned (the neutral element of +)
    """
    accumulator = 0
    for v in args:
        accumulator += v
    return accumulator

```

(continues on next page)

(continued from previous page)

```
@join_app
def fibonacci(n):
    if n == 0:
        return add()
    elif n == 1:
        return add(1)
    else:
        return add(fibonacci(n - 1), fibonacci(n - 2))

print(fibonacci(10).result())
```

Examples

Monte Carlo workflow

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

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

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

Thus, if N uniformly-distributed random points are dropped within the square, approximately $N\pi/4$ will be inside the circle.

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

The dependency chain looks like this:

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

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

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

    return (inside*4 / num_points)

# App that computes the mean of three values
```

(continues on next page)

(continued from previous page)

```

@python_app
def mean(a, b, c):
    return (a + b + c) / 3

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

# Compute the mean of the three estimates
mean_pi = mean(a, b, c)

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

```

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.

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'
        )
    ]
)

```

Local execution with pilot jobs

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

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

local_htex = Config(
    executors=[
        HighThroughputExecutor(
            label="htex_Local",
            worker_debug=True,
            cores_per_worker=1,
            provider=LocalProvider(
                channel=LocalChannel(),
                init_blocks=1,
                max_blocks=1,
            ),
        )
    ],
    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 {0} > {1}".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
```

(continues on next page)

(continued from previous page)

```
↪ ' % 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())
```

User guide

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.

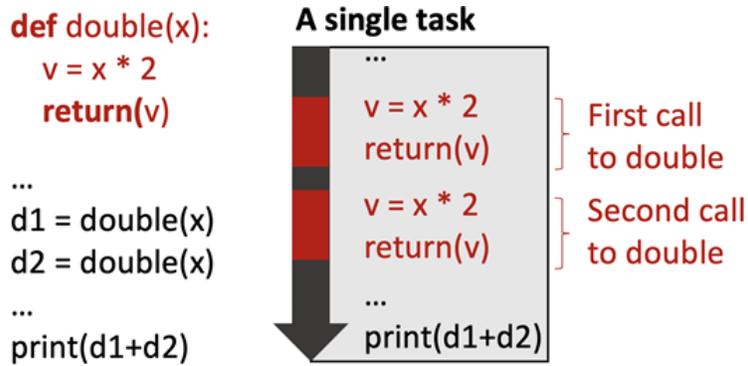
We aim in this section to provide a mental model of how Parsl programs behave. We discuss how Parsl programs create concurrent tasks, how tasks communicate, and the nature of the environment on which Parsl programs can perform operations. In each case, we compare and contrast the behavior of Python programs 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 `parsl.executors.HighThroughputExecutor` (HTEX).

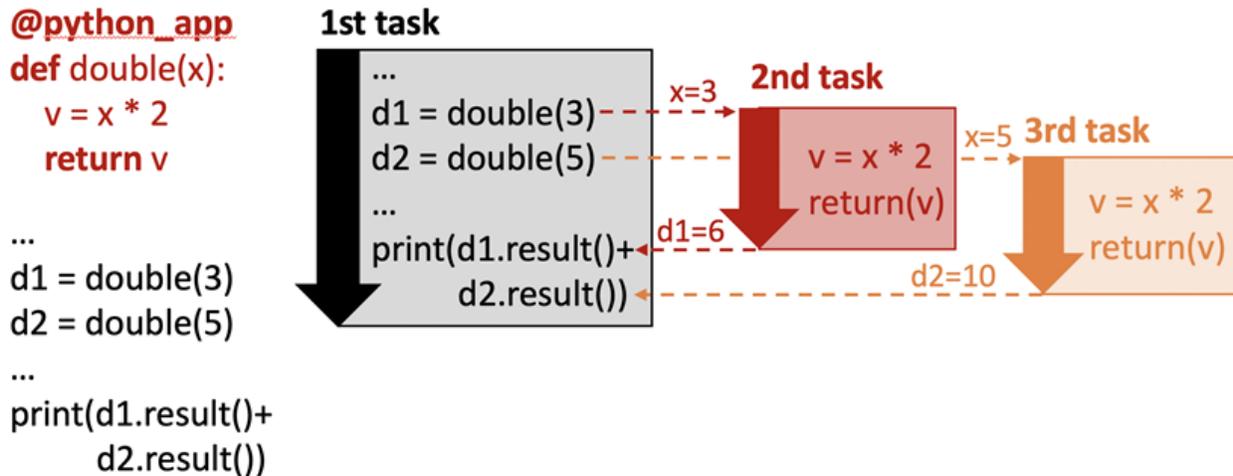
Parsl and Concurrency

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.

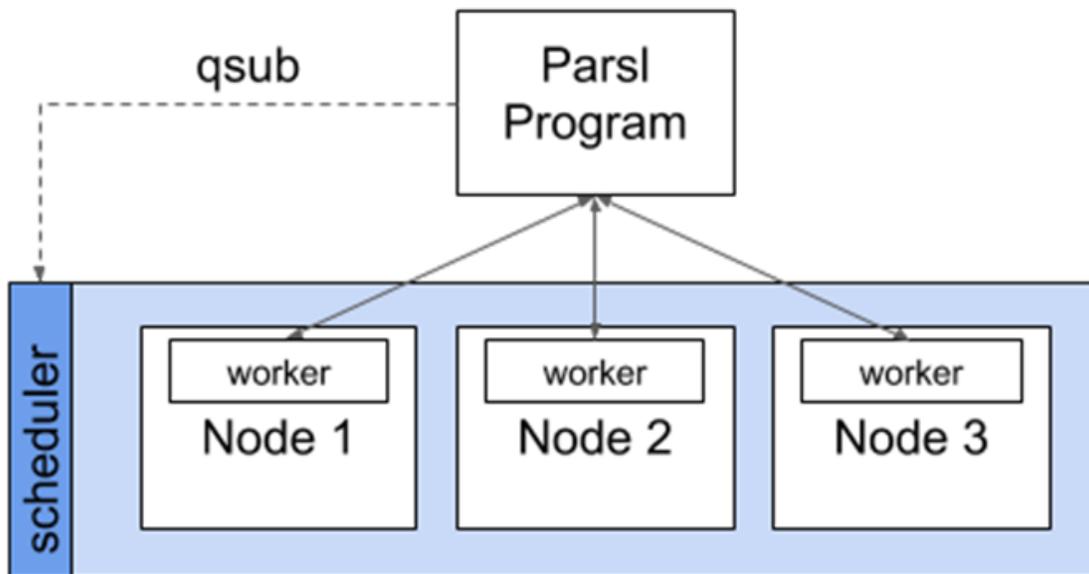
Parsl and Execution

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 `parsl.executors.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 `parsl.executors.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).

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

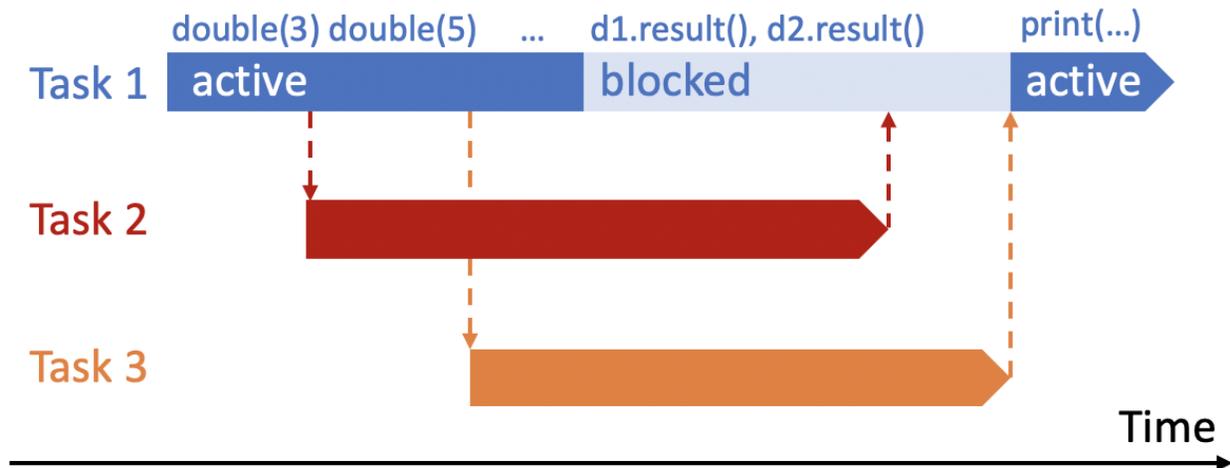
The figure above illustrates communication via parameter passing. The call `double(3)` to the app `double` in the main program creates a new task and passes the parameter value, 3, to that new task. When the task completes execution, its return value, 6, is returned to the main program. Similarly, the 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.



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 `parsl.executors.HighThroughputExecutor` which is representative of all Parsl executors except the `parsl.executors.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 `parsl.executors.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 `parsl.executors.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 `parsl.executors.ThreadPoolExecutor`.

Similarly, the same scoping rules apply to import statements, and thus the following program will run without errors with the `parsl.executors.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

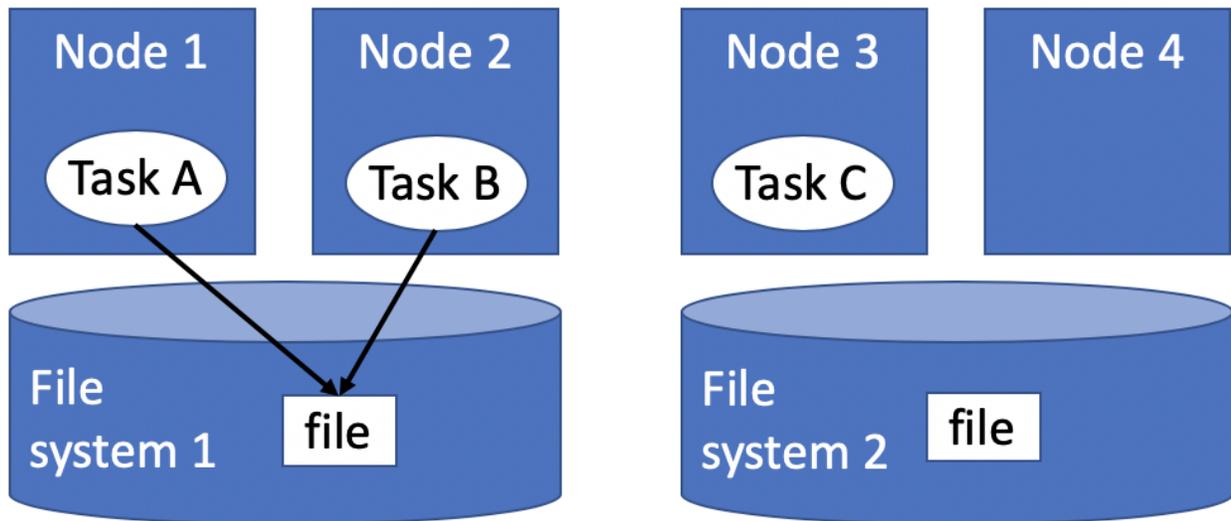
In a regular Python program the environment that is accessible to a Python program also includes the file system(s) of the computer on which it is executing. Thus in the following code, a value written to a file “answer.txt” in the current directory can be retrieved by reading the same file, and the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

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

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 `parsl.executors.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 ThreadPoolExecutor	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

Apps

An **App** defines a computation that will be executed asynchronously by Parsl. Apps are Python functions marked with a decorator which designates that the function will run asynchronously and cause it to return a `Future` instead of the result.

Apps can be one of three types of functions, each with their own type of decorator

- `@python_app`: Most Python functions
- `@bash_app`: A Python function which returns a command line program to execute
- `@join_app`: A function which launches one or more new Apps

The intricacies of Python and Bash apps are documented below. Join apps are documented in a later section (see *Join Apps*).

Python Apps

```
@python_app
def hello_world(name: str) -> str:
    return f'Hello, {name}!'

print(hello_world('user').result())
```

Python Apps run Python functions. The code inside a function marked by `@python_app` is what will be executed either locally or on a remote system.

Most functions can run without modification. Limitations on the content of the functions and their inputs/outputs are described below.

Rules for Function Contents

Parsl apps have access to less information from the script that defined them than functions run via Python's native multiprocessing libraries. The reason is that functions are executed on workers that lack access to the global variables in the script that defined them. Practically, this means

1. *Functions may need to re-import libraries.* Place the import statements that define functions or classes inside the function. Type annotations should not use libraries defined in the function.

```
import numpy as np

# BAD: Assumes library has been imported
@python_app
def linear_model(x: list[float] | np.ndarray, m: float, b: float):
    return np.multiply(x, m) + b

# GOOD: Function imports libraries on remote worker
@python_app
def linear_model(x: list[float] | 'np.ndarray', m: float, b: float):
    import numpy as np
    return np.multiply(x, m) + b
```

2. *Global variables are inaccessible.* Functions should not use variables defined outside the function. Likewise, do not assume that variables created inside the function are visible elsewhere.

```
# BAD: Uses global variables
global_var = {'a': 0}

@python_app
def counter_func(string: str, character: str = 'a'):
    global_var[character] += string.count(character) # `global_var` will not be
    ↪accessible

# GOOD
@python_app
def counter_func(string: str, character: str = 'a'):
    return {character: string.count(character)}

for ch, co in good_global('parsl', 'a').result().items():
    global_var[ch] += co
```

3. *Outputs are only available through return statements.* Parsl does not support generator functions (i.e., those which use yield statements) and any changes to input arguments will not be communicated.

```
# BAD: Assumes changes to inputs will be communicated
@python_app
def append_to_list(input_list: list, new_val):
    input_list.append(new_val)

# GOOD: Changes to inputs are returned
@python_app
def append_to_list(input_list: list, new_val) -> list:
```

(continues on next page)

(continued from previous page)

```
input_list.append(new_val)
return input_list
```

Functions from Modules

The above rules assume that the user is running the example code from a standalone script or Jupyter Notebook. Functions that are defined in an installed Python module do not need to abide by these guidelines, as they are sent to workers differently than functions defined locally within a script.

Directly convert a function from a library to a Python App by passing it as an argument to `python_app`:

```
from module import function
function_app = python_app(function)
```

`function_app` will act as Parsl App function of `function`.

It is also possible to create wrapped versions of functions, such as ones with pinned arguments. Parsl just requires first calling `update_wrapped()` with the wrapped function to include attributes from the original function (e.g., its name).

```
from functools import partial, update_wrapped
import numpy as np
my_max = partial(np.max, axis=0, keepdims=True)
my_max = update_wrapper(my_max, max) # Copy over the names
my_max_app = python_app(my_max)
```

The above example is equivalent to creating a new function (as below)

```
@python_app
def my_max_app(*args, **kwargs):
    import numpy as np
    return np.max(*args, keepdims=True, axis=0, **kwargs)
```

Inputs and Outputs

Python apps may be passed any Python type as an input and return any Python type, with a few exceptions. There are several classes of allowed types, each with different rules.

- *Python Objects*: Any Python object that can be saved with `pickle` or `dill` can be used as an import or output. All primitive types (e.g., floats, strings) are valid as are many complex types (e.g., numpy arrays).
- *Files*: Pass files as inputs as a `File` object. Parsl can transfer them to a remote system and update the `File` object with a new path. Access the new path with `File.filepath` attribute.

```
@python_app
def read_first_line(x: File):
    with open(x.filepath, 'r') as fp:
        return fp.readline()
```

Files can also be outputs of a function, but only through the `outputs` kwargs (described below).

- *Parsl Futures*. Functions can receive results from other Apps as Parsl Future objects. Parsl will establish a dependency on the App(s) which created the Future(s) and start executing as soon as the preceding ones complete.

```
@python_app
def capitalize(x: str):
    return x.upper()

input_file = File('text.txt')
first_line_future = read_first_line(input_file)
capital_future = capitalize(first_line_future)
print(capital_future.result())
```

See the section on [Futures](#) for more details.

Learn more about the types of data allowed in [the data section](#).

Note: Any changes to mutable input arguments will be ignored.

Special Keyword Arguments

Some keyword arguments to the Python function are treated differently by Parsl

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.

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

@python_app()
def reduce_app(inputs = ()):
    return sum(inputs)

map_futures = [map_app(x) for x in range(3)]
reduce_future = reduce_app(inputs=map_futures)

print(reduce_future.result()) # 0 + 1 * 2 + 2 * 2 = 6
```

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.

```
@python_app()
def write_app(message, outputs=()):
    """Write a single message to every file in outputs"""
    for path in outputs:
        with open(path, 'w') as fp:
            print(message, file=fp)

to_write = [
    File(Path(tmpdir) / 'output-0.txt'),
    File(Path(tmpdir) / 'output-1.txt')
]
```

(continues on next page)

(continued from previous page)

```
write_app('Hello!', outputs=to_write).result()
for path in to_write:
    with open(path) as fp:
        assert fp.read() == 'Hello!\n'
```

3. `walltime`: (int) This keyword argument places a limit on the app's runtime in seconds. If the walltime is exceeded, Parsl will raise an `parsl.app.errors.AppTimeout` exception.

Outputs

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.

Options for Python Apps

The `python_app()` decorator has a few options which controls how Parsl executes all tasks run with that application. For example, you can ensure that Parsl caches the results of the function and executes tasks on specific sites.

```
@python_app(cache=True, executors=['gpu'])
def expensive_gpu_function():
    # ...
    return
```

See the Parsl documentation for full details.

Limitations

To summarize, any Python function can be made a Python App with a few restrictions

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 if they are defined in script which starts Parsl.
3. Parsl uses dill and pickle to serialize Python objects to/from apps. Therefore, Parsl require that all input and output objects can be serialized by dill or pickle. See *Addressing SerializationError*.
4. STDOUT and STDERR produced by Python apps remotely are not captured.

Bash Apps

```
@bash_app
def echo(
    name: str,
    stdout=parsl.AUTO_LOGNAME # Requests Parsl to return the stdout
):
    return f'echo "Hello, {name}!"'

future = echo('user')
```

(continues on next page)

(continued from previous page)

```
future.result() # block until task has completed

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

A Parsl Bash app executes an external application by making a command-line execution. Parsl will execute the string returned by the function as a command-line script on a remote worker.

Rules for Function Contents

Bash Apps follow the same rules *as Python Apps*. For example, imports may need to be inside functions and global variables will be inaccessible.

Inputs and Outputs

Bash Apps can use the same kinds of inputs as Python Apps, but only communicate results with Files.

The Bash Apps, unlike Python Apps, can also return the content printed to the Standard Output and Error.

Special Keywords Arguments

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.

Outputs

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`.

Execution Options

Bash Apps have the same execution options (e.g., pinning to specific sites) as the Python Apps.

MPI Apps

Applications which employ MPI to span multiple nodes are a special case of Bash apps, and require special modification of Parsl's [execution environment](#) to function. Support for MPI applications is described in a [later section](#).

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.

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.
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.

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())
```

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 {}'.format(name)

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

Parsl uses the `dill` 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*.

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.

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
```

(continues on next page)

(continued from previous page)

```
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 these files are transferred depends on the scheme and the staging providers specified in the Parsl configuration.

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()],
        )
    ]
)
```

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/ar/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/ar/ARIN-STATS-FORMAT-CHANGE.txt')

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

(continues on next page)

(continued from previous page)

```
# 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()]
        )
    ]
)
```

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

(continues on next page)

(continued from previous page)

```
)
]
)
```

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
                ...
            )
        )
    ]
)
```

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.

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. `parsl.providers.LocalProvider`: The provider allows you to run locally on your laptop or workstation.
2. `parsl.providers.CobaltProvider`: This provider allows you to schedule resources via the Cobalt scheduler. **This provider is deprecated and will be removed by 2024.04.**
3. `parsl.providers.SlurmProvider`: This provider allows you to schedule resources via the Slurm scheduler.
4. `parsl.providers.CondorProvider`: This provider allows you to schedule resources via the Condor scheduler.
5. `parsl.providers.GridEngineProvider`: This provider allows you to schedule resources via the GridEngine scheduler.
6. `parsl.providers.TorqueProvider`: This provider allows you to schedule resources via the Torque scheduler.
7. `parsl.providers.AWSPProvider`: This provider allows you to provision and manage cloud nodes from Amazon Web Services.
8. `parsl.providers.GoogleCloudProvider`: This provider allows you to provision and manage cloud nodes from Google Cloud.
9. `parsl.providers.KubernetesProvider`: This provider allows you to provision and manage containers on a Kubernetes cluster.
10. `parsl.providers.AdHocProvider`: This provider allows you manage execution over a collection of nodes to form an ad-hoc cluster.

11. `parsl.providers.LSFProvider`: This provider allows you to schedule resources via IBM's LSF scheduler.

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 `Work Queue`. 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. `parsl.executors.ThreadPoolExecutor`: This executor supports multi-thread execution on local resources.
2. `parsl.executors.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. `parsl.executors.WorkQueueExecutor`: 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. `parsl.executors.taskvine.TaskVineExecutor`: This executor uses `TaskVine` as the execution backend. `TaskVine` scales up to tens of thousands of cores and actively uses local storage on compute nodes to offer a diverse array of performance-oriented features, including: smart caching and sharing common large files between tasks and compute nodes, reliable execution of tasks, dynamic resource sizing, automatic Python environment detection and sharing. 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.

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. `parsl.launchers.SrunLauncher`: Srun based launcher for Slurm based systems.
2. `parsl.launchers.AprunLauncher`: Aprun based launcher for Crays.
3. `parsl.launchers.SrunMPILauncher`: Launcher for launching MPI applications with Srun.

4. `parsl.launchers.GnuParallelLauncher`: Launcher using GNU parallel to launch workers across nodes and cores.
5. `parsl.launchers.MpiExecLauncher`: Uses Mpiexec to launch.
6. `parsl.launchers.SimpleLauncher`: The launcher default to a single worker launch.
7. `parsl.launchers.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()

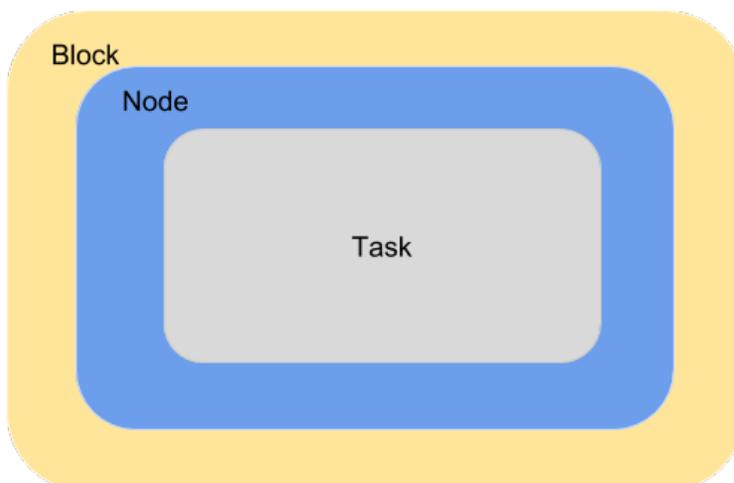
    def __call__(self, command, tasks_per_node, nodes_per_block):
        new_command="worker-wrapper {}".format(command)
        return self.srun_launcher(new_command, tasks_per_node, nodes_per_block)
```

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

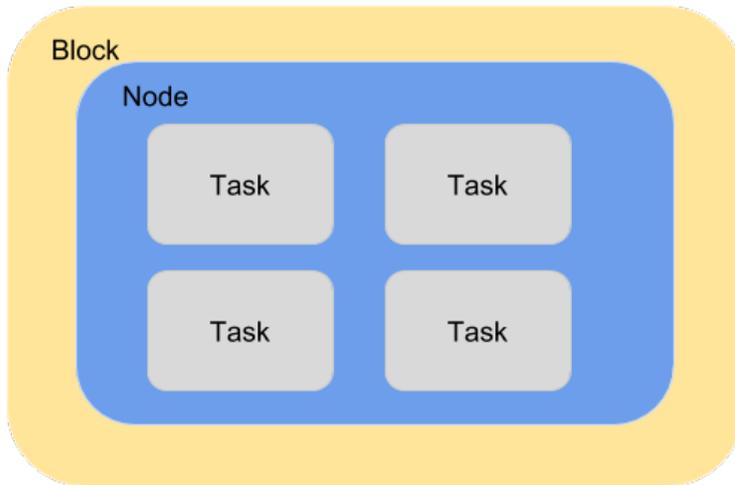
Parsl defines a resource abstraction called a *block* as the most basic unit of resources to be acquired from a provider. A block contains one or more nodes and maps to the different provider abstractions. In a cluster, a block corresponds to a single allocation request to a scheduler. In a cloud, a 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:

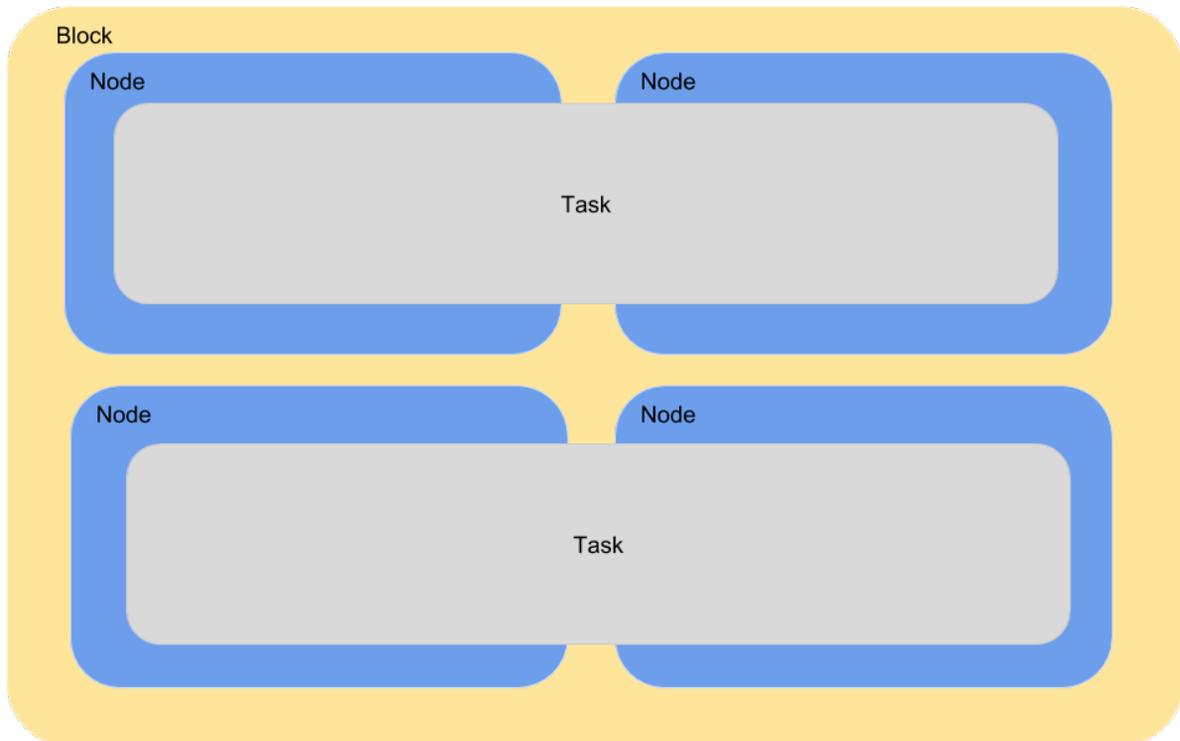


2. A single block with one node executing several tasks. This configuration is most suitable for single threaded apps running on multicore target systems. The number of tasks executed concurrently is proportional to the number

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 [MPI Apps](#) documentation page describes how to configure Parsl for this case.



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.

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. 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 a flow control system that monitors outstanding tasks and available compute capacity. This flow control monitor 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 `parsl.executors.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 $2 * \text{the total task capacity}$ (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
            )
        )
    ]
)
```

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.

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])

```

Encryption

Users can enable encryption for the `HighThroughputExecutor` by setting its `encrypted` initialization argument to `True`.

For example,

```

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

config = Config(
    executors=[
        HighThroughputExecutor(
            encrypted=True
        )
    ]
)

```

Under the hood, we use [CurveZMQ](#) to encrypt all communication channels between the executor and related nodes.

Encryption performance

`CurveZMQ` depends on `libzmq` and `libsodium`, which `pyzmq` (a Parsl dependency) includes as part of its installation via `pip`. This installation path should work on most systems, but users have reported significant performance degradation as a result.

If you experience a significant performance hit after enabling encryption, we recommend installing `pyzmq` with `conda`:

```
conda install conda-forge::pyzmq
```

Alternatively, you can install `libsodium`, then install `libzmq`, then build `pyzmq` from source:

```
pip3 install parsl --no-binary pyzmq
```

MPI Apps

MPI applications run multiple copies of a program that complete a single task by coordinating using messages passed within or across nodes. Starting MPI application requires invoking a “launcher” code (e.g., `mpiexec`) from one node with options that define how the copies of a program should be distributed to others. Parsl simplifies this by composing the “launcher” command from the resources specified at the time each app is invoked.

The broad strokes of a complete solution involves the following components:

1. **Configuring the *HighThroughputExecutor* with:**
 - `enable_mpi_mode=True`
2. Specify an MPI Launcher from one of the supported launchers (“`aprun`”, “`srun`”, “`mpiexec`”) for the *HighThroughputExecutor* with: `mpi_launcher="srun"`
3. Specify the provider that matches your cluster, (eg. user `SlurmProvider` for Slurm clusters)
4. Set the non-mpi launcher to *SingleNodeLauncher*
5. Specify resources required by the application via `resource_specification` as shown below:

```
# Define HighThroughputExecutor(enable_mpi_mode=True, mpi_launcher="mpiexec", ...)

@bash_app
def lammmps_mpi_application(infile: File, parsl_resource_specification: Dict):
    # PARSL_MPI_PREFIX will resolve to `mpiexec -n 4 -ppn 2 -hosts NODE001,NODE002`
    return f"${PARSL_MPI_PREFIX} lmp_mpi -in {infile.filepath}"

# Resources in terms of nodes and how ranks are to be distributed are set on a per app
# basis via the resource_spec dictionary.
resource_spec = {
    "num_nodes" = 2,
    "ranks_per_node" = 2,
    "num_ranks" = 4,
}
future = lammmps_mpi_application(File('in.file'), resource_specification=resource_spec)
```

HTEX and MPI Tasks

The *HighThroughputExecutor* (HTEX) is the default executor available through Parsl. Parsl Apps which invoke MPI code require MPI specific configuration such that:

1. All workers are started on the lead-node (mom-node in case of Crays)
2. Resource requirements of Apps are propagated to workers who provision the required number of nodes from within the batch job.

Configuring the Provider

Parsl must be configured to deploy workers on exactly one node per block. This part is simple. Instead of defining a launcher which will place an executor on each node in the block, simply use the *SingleNodeLauncher*. The MPI Launcher that the application will use is to be specified via `HighThroughputExecutor(mpi_launcher="LAUNCHER")`

It is also necessary to specify the desired number of blocks for the executor. Parsl cannot determine the number of blocks needed to run a set of MPI Tasks, so they must be set explicitly (see [Issue #1647](#)). The easiest route is to set the `max_blocks` and `min_blocks` of the provider to the desired number of blocks.

Configuring the Executor

Here are the steps for configuring the executor:

1. Set `HighThroughputExecutor(enable_mpi_mode=True)`
2. Set `HighThroughputExecutor(mpi_launcher="LAUNCHER")` to one from (“srun”, “aprun”, “mpiexec”)
3. Set the `max_workers` to the number of MPI Apps you expect to run per scheduler job (block).
4. Set `cores_per_worker=1e-6` to prevent HTEX from reducing the number of workers if you request more workers than cores.

Example Configuration

Here’s an example configuration which runs MPI tasks on ALCF’s Polaris Supercomputer

```
import parsl
from typing import Dict
from parsl.config import Config

# PBSPro is the right provider for Polaris:
from parsl.providers import PBSProProvider
# The high throughput executor is for scaling to HPC systems:
from parsl.executors import HighThroughputExecutor
# address_by_interface is needed for the HighThroughputExecutor:
from parsl.addresses import address_by_interface
# For checkpointing:
from parsl.utils import get_all_checkpoints

# Adjust your user-specific options here:
# run_dir="/lus/grand/projects/yourproject/yourrundir/"

user_opts = {
    "worker_init": "module load conda; conda activate parsl_mpi_py310",
    "scheduler_options": "#PBS -l filesystems=home:eagle:grand\n#PBS -l place=scatter" ,
    "account": SET_YOUR_ALCF_ALLOCATION_HERE,
    "queue": "debug-scaling",
    "walltime": "1:00:00",
    "nodes_per_block": 8,
    "available_accelerators": 4, # Each Polaris node has 4 GPUs, setting this ensures
    ↪ one worker per GPU
```

(continues on next page)

(continued from previous page)

```

    "cores_per_worker": 8, # this will set the number of cpu hardware threads per worker.
}

config = Config(
    executors=[
        HighThroughputExecutor(
            label="htex",
            enable_mpi_mode=True,
            mpi_launcher="mpiexec",
            cores_per_worker=user_opts["cores_per_worker"],
            address=address_by_interface("bond0"),
            provider=PBSProProvider(
                account=user_opts["account"],
                queue=user_opts["queue"],
                # PBS directives (header lines): for array jobs pass '-J' option
                scheduler_options=user_opts["scheduler_options"],
                # Command to be run before starting a worker, such as:
                worker_init=user_opts["worker_init"],
                # number of compute nodes allocated for each block
                nodes_per_block=user_opts["nodes_per_block"],
                init_blocks=1,
                min_blocks=0,
                max_blocks=1, # Can increase more to have more parallel jobs
                walltime=user_opts["walltime"]
            ),
        ),
    ],
)

```

Writing MPI-Compatible Apps

In MPI mode, the *HighThroughputExecutor* can execute both Python or Bash Apps which invokes the MPI application. However, it is important to note that Python Apps that directly use `mpi4py` is not supported.

For multi-node MPI applications, especially when running multiple applications within a single batch job, it is important to specify the resource requirements for the app so that the Parsl worker can provision the appropriate resources before the application starts. For eg, your Parsl script might contain a molecular dynamics application that requires 8 ranks over 1 node for certain inputs and 32 ranks over 4 nodes for some depending on the size of the molecules being simulated. By specifying resources via `resource_specification`, parsl workers will provision the requested resources and then compose MPI launch command prefixes (Eg: `mpiexec -n <ranks> -ppn <ranks_per_node> -hosts <node1..nodeN>`). These launch command prefixes are shared with the app via environment variables.

```

@bash_app
def echo_hello(n: int, stderr='std.err', stdout='std.out', parsl_resource_specification: Dict):
    return f'$PARSL_MPI_PREFIX hostname'

# The following app will echo the hostname from several MPI ranks
# Alternatively, you could also use the resource_specification to compose a launch
# command using env vars set by Parsl from the resource_specification like this:
@bash_app
def echo_hostname(n: int, stderr='std.err', stdout='std.out', parsl_resource_

```

(continues on next page)

(continued from previous page)

```

↪specification: Dict):
    total_ranks = os.environ("")
    return f'aprun -N $PARSL_RANKS_PER_NODE -n {total_ranks} /bin/hostname'

```

All valid key-value pairs set in the `resource_specification` are exported to the application via env vars, for eg. `parsl_resource_specification = {'RANKS_PER_NODE': 4}` `` will set the env var ```PARSL_RANKS_PER_NODE`

However, the following options are **required** for MPI applications :

```

resource_specification = {
    'num_nodes': <int>,          # Number of nodes required for the application instance
    'ranks_per_node': <int>,    # Number of ranks / application elements to be launched per_
↪node
    'num_ranks': <int>,        # Number of ranks in total
}

# The above are made available in the worker env vars:
# echo $PARSL_NUM_NODES, $PARSL_RANKS_PER_NODE, $PARSL_NUM_RANKS

```

When the above are supplied, the following launch command prefixes are set:

```

PARSL_MPIEXEC_PREFIX: mpiexec launch command which works for a large number of batch_
↪systems especially PBS systems
PARSL_SRUN_PREFIX: srun launch command for Slurm based clusters
PARSL_APRUN_PREFIX: aprun launch command prefix for some Cray machines
PARSL_MPI_PREFIX: Parsl sets the MPI prefix to match the mpi_launcher specified to_
↪`HighThroughputExecutor`
PARSL_MPI_NODELIST: List of assigned nodes separated by commas (Eg, NODE1,NODE2)
PARSL_WORKER_POOL_ID: Alphanumeric string identifier for the worker pool
PARSL_WORKER_BLOCK_ID: Batch job ID that the worker belongs to

```

Example Application: CosmicTagger

TODO: Blurb about what CosmicTagger does CosmicTagger implements models and training utilities to train convolutional networks to separate cosmic pixels, background pixels, and neutrino pixels in a neutrinos dataset. There are several variations. A detailed description of the code can be found in:

Cosmic Background Removal with Deep Neural Networks in SBND

Cosmic Background Removal with Deep Neural Networks in SBND This network is implemented in both PyTorch and TensorFlow. To select between the networks, you can use the `--framework` parameter. It accepts either `tensorflow` or `torch`. The model is available in a development version with sparse convolutions in the `torch` framework.

This example is broken down into three components. First, configure the Executor for Polaris at ALCF. The configuration will use the *PBSProProvider* to connect to the batch scheduler. With the goal of running MPI applications, we set the

```

import parsl
from typing import Dict
from parsl.config import Config

# PBSPro is the right provider for Polaris:

```

(continues on next page)

(continued from previous page)

```

from parsl.providers import PBSProProvider
# The high throughput executor is for scaling to HPC systems:
from parsl.executors import HighThroughputExecutor
# address_by_interface is needed for the HighThroughputExecutor:
from parsl.addresses import address_by_interface

user_opts = {
    # Make sure to setup a conda environment before using this config
    "worker_init": "module load conda; conda activate parsl_mpi_py310",
    "scheduler_options": "#PBS -l filesystems=home:eagle:grand\n#PBS -l place=scatter" ,
    "account": <SET_YOUR_ALLOCATION>,
    "queue": "debug-scaling",
    "walltime": "1:00:00",
    "nodes_per_block": 8,
    "available_accelerators": 4, # Each Polaris node has 4 GPUs, setting this ensures
    ↪one worker per GPU
    "cores_per_worker": 8, # this will set the number of cpu hardware threads per worker.
}

config = Config(
    executors=[
        HighThroughputExecutor(
            label="htex",
            enable_mpi_mode=True,
            mpi_launcher="mpiexec",
            cores_per_worker=user_opts["cores_per_worker"],
            address=address_by_interface("bond0"),
            provider=PBSProProvider(
                account=user_opts["account"],
                queue=user_opts["queue"],
                # PBS directives (header lines): for array jobs pass '-J' option
                scheduler_options=user_opts["scheduler_options"],
                # Command to be run before starting a worker, such as:
                worker_init=user_opts["worker_init"],
                # number of compute nodes allocated for each block
                nodes_per_block=user_opts["nodes_per_block"],
                init_blocks=1,
                min_blocks=0,
                max_blocks=1, # Can increase more to have more parallel jobs
                walltime=user_opts["walltime"]
            ),
        ),
    ],
)

```

Next we define the CosmicTagger MPI application. TODO: Ask Khalid for help.

```

@parsl.bash_app
def cosmic_tagger(workdir: str,
                  datatype: str = "float32",
                  batchsize: int = 8,
                  framework: str = "torch",

```

(continues on next page)

(continued from previous page)

```

        iterations: int = 500,
        trial: int = 2,
        stdout=parsl.AUTO_LOGNAME,
        stderr=parsl.AUTO_LOGNAME,
        parsl_resource_specification:Dict={})):
NRANKS = parsl_resource_specification['num_ranks']

return f"""
module purge
module use /soft/modulefiles/
module load conda/2023-10-04
conda activate

echo "PARSL_MPI_PREFIX : $PARSL_MPI_PREFIX"

$PARSL_MPI_PREFIX --cpu-bind numa \
  python {workdir}/bin/exec.py --config-name a21 \
    run.id=run_plrs_ParslDemo_g${NRANKS}_{datatype}_b{batchsize}_{framework}_i
↪{iterations}_T{trial} \
    run.compute_mode=GPU \
    run.distributed=True \
    framework={framework} \
    run.minibatch_size={batchsize} \
    run.precision={datatype} \
    mode.optimizer.loss_balance_scheme=light \
    run.iterations={iterations}
"""

```

In this example, we run a simple test that does an exploration over the batchsize parameter while launching the application over 2-4 nodes.

```

def run_cosmic_tagger():
    futures = {}
    for num_nodes in [2, 4]:
        for batchsize in [2, 4, 8]:

            parsl_res_spec = {"num_nodes": num_nodes,
                             "num_tasks": num_nodes * 4,
                             "ranks_per_node": 4}
            future = cosmic_tagger(workdir="/home/yadunand/CosmicTagger",
                                   datatype="float32",
                                   batchsize=str(batchsize),
                                   parsl_resource_specification=parsl_res_spec)

            print(f"Stdout : {future.stdout}")
            print(f"Stderr : {future.stderr}")
            futures[(num_nodes, batchsize)] = future

    for key in futures:
        print(f"Got result for {key}: {futures[key].result()}")

```

(continues on next page)

```
run_cosmic_tagger()
```

Limitations

Support for MPI tasks in HTEX is limited. It is designed for running many multi-node MPI applications within a single batch job.

1. MPI tasks may not span across nodes from more than one block.
2. Parsl does not correctly determine the number of execution slots per block ([Issue #1647](#))
3. The executor uses a Python process per task, which can use a lot of memory ([Issue #2264](#))

Error handling

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

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)
```

(continues on next page)

(continued from previous page)

```
# 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.')
```

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)
```

More specific retry handling can be specified using retry handlers, documented below.

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 ----->
```

Retry handlers

The basic parsl retry mechanism keeps a count of the number of times a task has been (re)tried, and will continue retrying that task until the configured retry limit is reached.

Retry handlers generalize this to allow more expressive retry handling: parsl keeps a retry cost for a task, and the task will be retried until the configured retry limit is reached. Instead of the cost being 1 for each failure, user-supplied code can examine the failure and compute a custom cost.

This allows user knowledge about failures to influence the retry mechanism: an exception which is almost definitely a non-recoverable failure (for example, due to bad parameters) can be given a high retry cost (so that it will not be retried many times, or at all), and exceptions which are likely to be transient (for example, where a worker node has died) can be given a low retry cost so they will be retried many times.

A retry handler can be specified in the parsl configuration like this:

```
Config(
    retries=2,
    retry_handler=example_retry_handler
)
```

`example_retry_handler` should be a function defined by the user that will compute the retry cost for a particular failure, given some information about the failure.

For example, the following handler will give a cost of 1 to all exceptions, except when a bash app exits with unix exitcode 9, in which case the cost will be 100. This will have the effect that retries will happen as normal for most errors, but the bash app can indicate that there is little point in retrying by exiting with exitcode 9.

```
def example_retry_handler(exception, task_record):
    if isinstance(exception, BashExitFailure) and exception.exitcode == 9:
        return 100
    else:
        return 1
```

The retry handler is given two parameters: the exception from execution, and the parsl internal `task_record`. The task record contains details such as the app name, parameters and executor.

If a retry handler raises an exception itself, then the task will be aborted and no further tries will be attempted.

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.

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 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.

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)])
```

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 `parsl.executors.HighThroughputExecutor` to submit tasks from a login node (`parsl.channels.LocalChannel`). It requests an allocation of 128 nodes, deploying 1 worker for each of the 56 cores per node, from the normal partition. To limit network connections to just the internal network the config specifies the address used by the infiniband interface with `address_by_interface('ib0')`

```
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_interface

config = Config(
    executors=[
        HighThroughputExecutor(
            label="frontera_htex",
            address=address_by_interface('ib0'),
            max_workers_per_node=56,
            provider=SlurmProvider(
                channel=LocalChannel(),
                nodes_per_block=128,
                init_blocks=1,
                partition='normal',
                launcher=SrunLauncher(),
            ),
        ),
    ],
)
```

Configuration How-To and Examples:

- *Configuration*
 - *Creating and Using Config Objects*
 - *How to Configure*
 - *Heterogeneous Resources*
 - *Accelerators*
 - *Multi-Threaded Applications*
 - *Ad-Hoc Clusters*
 - *Amazon Web Services*
 - *ASPIRE 1 (NSCC)*

- *Illinois Campus Cluster (UIUC)*
- *Bridges (PSC)*
- *CC-IN2P3*
- *CCL (Notre Dame, TaskVine)*
- *Expanse (SDSC)*
- *Perlmutter (NERSC)*
- *Frontera (TACC)*
- *Kubernetes Clusters*
- *Midway (RCC, UChicago)*
- *Open Science Grid*
- *Polaris (ALCF)*
- *Stampede2 (TACC)*
- *Summit (ORNL)*
- *TOSS3 (LLNL)*
- *Further help*

Creating and Using Config Objects

Config objects are loaded to define the “Data Flow Kernel” (DFK) that will manage tasks. All Parsl applications start by creating or importing a configuration then calling the load function.

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

with parsl.load(config):
```

The load statement can happen after Apps are defined but must occur before tasks are started. Loading the Config object within context manager like `with` is recommended for implicit cleaning of DFK on exiting the context manager

The *Config* object may not be used again after loaded. Consider a configuration function if the application will shut down and re-launch the DFK.

```
from parsl.config import Config
import parsl

def make_config() -> Config:
    return Config(...)

with parsl.load(make_config()):
    # Your workflow here
parsl.clear() # Stops Parsl
with parsl.load(make_config()): # Re-launches with a fresh configuration
    # Your workflow here
```

How to Configure

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

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"> <code>parsl.executors.HighThroughputExecutor</code> <code>parsl.executors.ThreadPoolExecutor</code> <code>parsl.executors.WorkQueueExecutor</code> <code>parsl.executors.taskvine.TaskVineExecutor</code> 	<code>parsl.providers.LocalProvider</code>
Amazon Web Services	<ul style="list-style-type: none"> <code>parsl.executors.HighThroughputExecutor</code> 	<code>parsl.providers.AWSProvider</code>
Google Cloud	<ul style="list-style-type: none"> <code>parsl.executors.HighThroughputExecutor</code> 	<code>parsl.providers.GoogleCloudProvider</code>
Slurm based system	<ul style="list-style-type: none"> <code>parsl.executors.HighThroughputExecutor</code> <code>parsl.executors.WorkQueueExecutor</code> <code>parsl.executors.taskvine.TaskVineExecutor</code> 	<code>parsl.providers.SlurmProvider</code>
Torque/PBS based system	<ul style="list-style-type: none"> <code>parsl.executors.HighThroughputExecutor</code> <code>parsl.executors.WorkQueueExecutor</code> 	<code>parsl.providers.TorqueProvider</code>
Cobalt based system	<ul style="list-style-type: none"> <code>parsl.executors.HighThroughputExecutor</code> <code>parsl.executors.WorkQueueExecutor</code> 	<code>parsl.providers.CobaltProvider</code>
GridEngine based system	<ul style="list-style-type: none"> <code>parsl.executors.HighThroughputExecutor</code> <code>parsl.executors.WorkQueueExecutor</code> 	<code>parsl.providers.GridEngineProvider</code>
Condor based cluster or grid	<ul style="list-style-type: none"> <code>parsl.executors.HighThroughputExecutor</code> <code>parsl.executors.WorkQueueExecutor</code> <code>parsl.executors.taskvine.TaskVineExecutor</code> 	<code>parsl.providers.CondorProvider</code>
Kubernetes cluster	<ul style="list-style-type: none"> <code>parsl.executors.HighThroughputExecutor</code> 	<code>parsl.providers.KubernetesProvider</code>

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>*parsl.executors.ThreadPoolExecutor</i>	1 (Only local)	Any
<i>parsl.executors.HighThroughputExecutor</i>	<=2000	Task duration(s)/#nodes >= 0.01 longer tasks needed at higher scale
<i>parsl.executors.WorkQueueExecutor</i>	<=1000 ^{†0}	10s+
<i>parsl.executors.taskvine.TaskVineExecutor</i>	<=1000 ^{‡0}	10s+

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>parsl.launchers.SingleNodeLauncher</i> <i>parsl.launchers.SimpleLauncher</i>
Aprun based systems	Any	<ul style="list-style-type: none"> <i>parsl.launchers.AprunLauncher</i>

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

^{†0} The maximum number of nodes tested for the *parsl.executors.WorkQueueExecutor* is 10,000 GPU cores and 20,000 CPU cores.

^{‡0} The maximum number of nodes tested for the *parsl.executors.taskvine.TaskVineExecutor* is 10,000 GPU cores and 20,000 CPU cores.

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

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

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

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

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 `parsl.executors.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 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=[
```

(continues on next page)

(continued from previous page)

```

    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.

Accelerators

Many modern clusters provide multiple accelerators per compute node, yet many applications are best suited to using a single accelerator per task. Parsl supports pinning each worker to different accelerators using the `available_accelerators` option of the `HighThroughputExecutor`. Provide either the number of executors (Parsl will assume they are named in integers starting from zero) or a list of the names of the accelerators available on the node.

```

local_config = Config(
    executors=[
        HighThroughputExecutor(
            label="htex_Local",
            worker_debug=True,
            available_accelerators=2,
            provider=LocalProvider(
                channel=LocalChannel(),
                init_blocks=1,
                max_blocks=1,
            ),
        )
    ],
    strategy='none',
)

```

Multi-Threaded Applications

Workflows which launch multiple workers on a single node which perform multi-threaded tasks (e.g., NumPy, Tensorflow operations) may run into thread contention issues. Each worker may try to use the same hardware threads, which leads to performance penalties. Use the `cpu_affinity` feature of the *HighThroughputExecutor* to assign workers to specific threads. Users can pin threads to workers either with a strategy method or an explicit list.

The strategy methods will auto assign all detected hardware threads to workers. Allowed strategies that can be assigned to `cpu_affinity` are `block`, `block-reverse`, and `alternating`. The `block` method pins threads to workers in sequential order (ex: 4 threads are grouped (0, 1) and (2, 3) on two workers); `block-reverse` pins threads in reverse sequential order (ex: (3, 2) and (1, 0)); and `alternating` alternates threads among workers (ex: (0, 2) and (1, 3)).

Select the best blocking strategy for processor’s cache hierarchy (choose `alternating` if in doubt) to ensure workers to not compete for cores.

```
local_config = Config(
    executors=[
        HighThroughputExecutor(
            label="htex_Local",
            worker_debug=True,
            cpu_affinity='alternating',
            provider=LocalProvider(
                channel=LocalChannel(),
                init_blocks=1,
                max_blocks=1,
            ),
        )
    ],
    strategy='none',
)
```

Users can also use `cpu_affinity` to assign explicitly threads to workers with a string that has the format of `cpu_affinity="list:<worker1_threads>:<worker2_threads>:<worker3_threads>"`.

Each worker’s threads can be specified as a comma separated list or a hyphenated range: `thread1, thread2, thread3` or `thread_start-thread_end`.

An example for 12 workers on a node with 208 threads is:

```
cpu_affinity="list:0-7,104-111:8-15,112-119:16-23,120-127:24-31,128-135:32-39,136-143:40-
↪47,144-151:52-59,156-163:60-67,164-171:68-75,172-179:76-83,180-187:84-91,188-195:92-99,
↪196-203"
```

This example assigns 16 threads each to 12 workers. Note that in this example there are threads that are skipped. If a thread is not explicitly assigned to a worker, it will be left idle. The number of thread “ranks” (colon separated thread lists/ranges) must match the total number of workers on the node; otherwise an exception will be raised.

Thread affinity is accomplished in two ways. Each worker first sets the affinity for the Python process using the `affinity mask`, which may not be available on all operating systems. It then sets environment variables to control `OpenMP thread affinity` so that any subprocesses launched by a worker which use OpenMP know which processors are valid. These include `OMP_NUM_THREADS`, `GOMP_COMP_AFFINITY`, and `KMP_THREAD_AFFINITY`.

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 use the `parsl.providers.AdHocProvider` with an `parsl.channels.SSHChannel` to each node. An example configuration follows.

```

from parsl.providers import AdHocProvider
from parsl.channels import SSHChannel
from parsl.executors import HighThroughputExecutor
from parsl.config import Config
from typing import Any, Dict

user_opts: Dict[str, Dict[str, Any]]
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_per_node=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 `parsl.executors.HighThroughputExecutor`

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 `parsl.providers.AWSProvider` to connect to AWS.

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

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

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

```

    )
],
)

```

ASPIRE 1 (NSCC)

The following snippet shows an example configuration for accessing NSCC's **ASPIRE 1** supercomputer. This example uses the `parsl.executors.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_per_node=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,
)

```

(continues on next page)

(continued from previous page)

```

    checkpoint_mode='task_exit'
)

```

Illinois Campus Cluster (UIUC)



The following snippet shows an example configuration for executing on the Illinois Campus Cluster. The configuration assumes the user is running on a login node and uses the `parsl.providers.SlurmProvider` to interface with the scheduler, and uses the `parsl.launchers.SrunLauncher` to launch workers.

```

from parsl.config import Config
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 the Campus Cluster at UIUC. Each job submitted to the scheduler will request 2 nodes,
↳for 10 minutes.
"""

config = Config(
    executors=[
        HighThroughputExecutor(
            label="CC_htex",
            worker_debug=False,
            cores_per_worker=16.0, # each worker uses a full node
            provider=SlurmProvider(
                partition='secondary-fdr', # partition
                nodes_per_block=2, # number of nodes
                init_blocks=1,
                max_blocks=1,
                scheduler_options='',
                cmd_timeout=60,
                walltime='00:10:00',
                launcher=SrunLauncher(),
                worker_init='conda activate envParsl', # requires conda environment.↳
            ),
        ),
    ],
)

```

Bridges (PSC)



The following snippet shows an example configuration for executing on the Bridges supercomputer at the Pittsburgh Supercomputing Center. The configuration assumes the user is running on a login node and uses the `parsl.providers.SlurmProvider` to interface with the scheduler, and uses the `parsl.launchers.SrunLauncher` to launch workers.

```

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

""" 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',
            address=address_by_interface('ens3f0'),
            max_workers_per_node=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',

```

(continues on next page)

(continued from previous page)

```

        # Slurm scheduler on Cori can be slow at times,
        # increase the command timeouts
        cmd_timeout=120,
    ),
]
)

```

CC-IN2P3



The snippet below shows an example configuration for executing from a login node on IN2P3's Computing Centre. The configuration uses the `parsl.providers.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 `parsl.providers.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_per_node=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
            )
        )
    ]
)

```

(continues on next page)

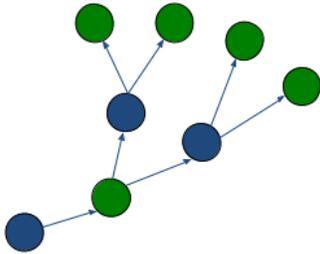
(continued from previous page)

```

    ),
  ],
)

```

CCL (Notre Dame, TaskVine)



TaskVine

To utilize TaskVine 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 TaskVine with the Parsl library.

The following snippet shows an example configuration for using the Parsl/TaskVine executor to run applications on the local machine. This examples uses the `parsl.executors.taskvine.TaskVineExecutor` to schedule tasks, and a local worker will be started automatically. For more information on using TaskVine, including configurations for remote execution, visit the [TaskVine/Parsl documentation online](#).

```

from parsl.config import Config
from parsl.executors.taskvine import TaskVineExecutor
from parsl.executors.taskvine import TaskVineManagerConfig
import uuid

config = Config(
    executors=[
        TaskVineExecutor(
            label="parsl-vine-example",

            # If a project_name is given, then TaskVine will periodically
            # report its status and performance back to the global TaskVine catalog,
            # which can be viewed here: http://ccl.cse.nd.edu/software/taskvine/status

            # To disable status reporting, comment out the project_name.
            manager_config=TaskVineManagerConfig(project_name="parsl-vine-" + str(uuid.
↪uid4()))),
    ]
)

```

(continues on next page)

(continued from previous page)

```
]
)
```

TaskVine’s predecessor, WorkQueue, may continue to be used with Parsl. For more information on using WorkQueue visit the [CCTools documentation](#) online.

Expanse (SDSC)



The following snippet shows an example configuration for executing remotely on San Diego Supercomputer Center’s **Expanse** supercomputer. The example is designed to be executed on the login nodes, using the `parsl.providers.SlurmProvider` to interface with the Slurm scheduler used by Comet and the `parsl.launchers.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='Expanse_CPU_Multinode',
            max_workers_per_node=32,
            provider=SlurmProvider(
                'compute',
                account='YOUR_ALLOCATION_ON_EXPANSE',
                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:
```

(continues on next page)

(continued from previous page)

```

        # 'module load Anaconda; source activate parsl_env'.
        worker_init='',
        walltime='01:00:00',
        init_blocks=1,
        max_blocks=1,
        nodes_per_block=2,
    ),
)
]
)

```

Perlmutter (NERSC)

NERSC provides documentation on how to use Parsl on Perlmutter.

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 `parsl.providers.SlurmProvider` to interface with the scheduler, and uses the `parsl.launchers.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=[

```

(continues on next page)

(continued from previous page)

```

HighThroughputExecutor(
    label='frontera_htex',
    max_workers_per_node=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(),
    ),
),
],
)

```

Kubernetes Clusters



kubernetes

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

```

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

```

(continues on next page)

```
config = Config(
    executors=[
        HighThroughputExecutor(
            label='kube-htex',
            cores_per_worker=1,
            max_workers_per_node=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,
            ),
        ],
    )
```

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 `parsl.providers.SlurmProvider` to interface with the scheduler, and uses the `parsl.launchers.SrunLauncher` to launch workers.

```

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

config = Config(
    executors=[
        HighThroughputExecutor(
            label='Midway_HTEX_multinode',
            address=address_by_interface('bond0'),
            worker_debug=False,
            max_workers_per_node=2,
            provider=SlurmProvider(
                'YOUR_PARTITION', # Partition name, e.g 'broadwl'
                launcher=SrunLauncher(),
                nodes_per_block=2,
                init_blocks=1,
                min_blocks=1,
                max_blocks=1,
                # string to prepend to #SBATCH blocks in the submit
                # script to the scheduler eg: '#SBATCH --constraint=knl,quad,cache'
                scheduler_options='',
                # Command to be run before starting a worker, such as:
                # 'module load Anaconda; source activate parsl_env'.
                worker_init='',
                walltime='00:30:00'
            ),
        ),
    ],

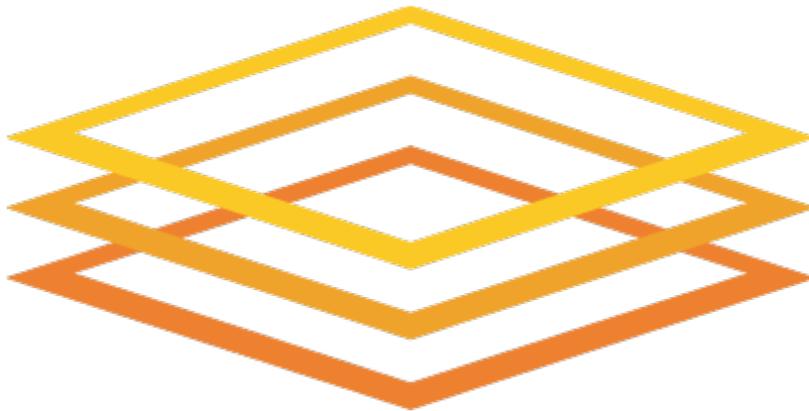
```

(continues on next page)

(continued from previous page)

```
)  
    ],  
)
```

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 *parsl.providers.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_per_node=1,  
            provider=CondorProvider(  
                nodes_per_block=1,  
                init_blocks=4,  
                max_blocks=4,  
                # This scheduler option string ensures that the compute nodes provisioned
```

(continues on next page)

(continued from previous page)

```

        # 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'.
        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)
)
]
)

```

Polaris (ALCF)

ALCF provides documentation on how to use Parsl on Polaris.

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
from parsl.addresses import address_by_interface

config = Config(
    executors=[
        HighThroughputExecutor(
            label='Stampede2_HTEX',
            address=address_by_interface('em3'),
            max_workers_per_node=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='cee5ca0-89a9-11e7-a97f-22000a92523b',
        endpoint_path='/',
        local_path='/')
    ])
    ],
)

```

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 `parsl.providers.LSFProvider` to provision compute nodes from the LSF cluster scheduler and the `parsl.launchers.JsrunLauncher` to launch workers across the compute nodes.

```

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

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

from parsl.addresses import address_by_interface

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

```

(continues on next page)

(continued from previous page)

```

    ),
)
],
)

```

TOSS3 (LLNL)



The following snippet shows an example configuration for executing on one of LLNL's **TOSS3** machines, such as Quartz, Ruby, Topaz, Jade, or Magma. This example uses the `parsl.executors.FluxExecutor` and connects to Slurm using the `parsl.providers.SlurmProvider`. This configuration assumes that the script is being executed on the login nodes of one of the machines.

```

from parsl.config import Config
from parsl.executors import FluxExecutor
from parsl.providers import SlurmProvider
from parsl.launchers import SrunLauncher

config = Config(
    executors=[
        FluxExecutor(
            provider=SlurmProvider(
                partition="YOUR_PARTITION", # e.g. "pbatch", "pdebug"
                account="YOUR_ACCOUNT",
                launcher=SrunLauncher(overrides="--mpibind=off"),

```

(continues on next page)

(continued from previous page)

```

        nodes_per_block=1,
        init_blocks=1,
        min_blocks=1,
        max_blocks=1,
        walltime="00:30:00",
        # string to prepend to #SBATCH blocks in the submit
        # script to the scheduler, e.g.: '#SBATCH -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,
    ),
)
]
)

```

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)

```

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.

Monitoring configuration

Parsl monitoring is only supported with the `parsl.executors.HighThroughputExecutor`.

The following example shows how to enable monitoring in the Parsl configuration. Here the `parsl.monitoring.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_per_node=4,
            address=address_by_hostname(),
        )
    ],
    monitoring=MonitoringHub(
        hub_address=address_by_hostname(),
        hub_port=55055,
        monitoring_debug=False,
        resource_monitoring_interval=10,
    ),
    strategy='none'
)

```

Visualization

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

```
$ pip install 'parsl[monitoring,visualization]'
```

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 `runinfo` 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:

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	21 0	Link
test_monitor_on_fail.py	2019-06-13 11:02:02	yadu	Completed	0:00:02	0 1	Link

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.

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:

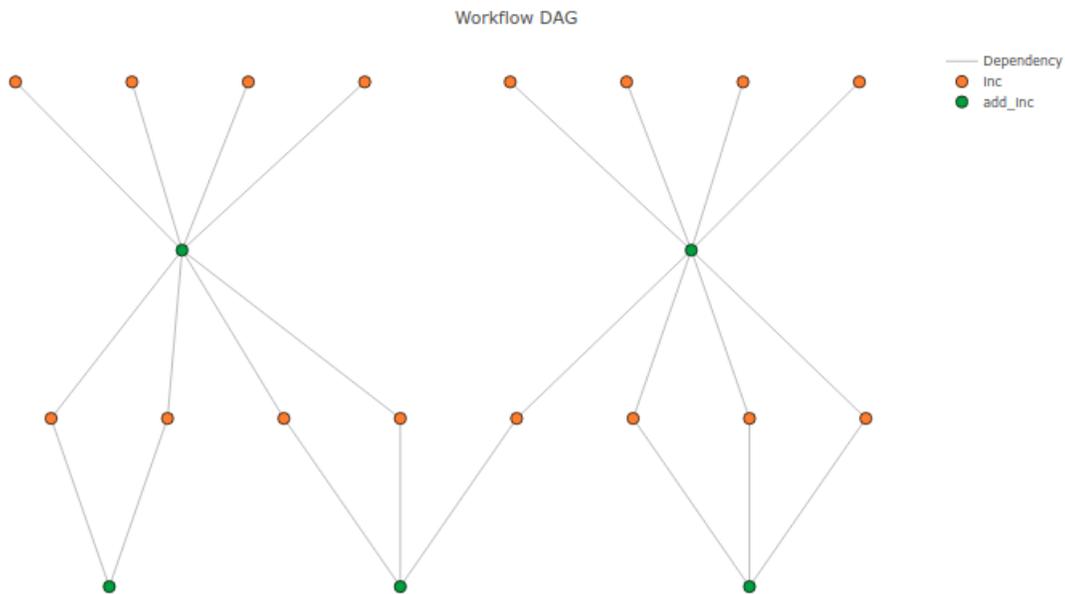
- Workflow DAG - with apps differentiated by colors: This visualization is useful to visually inspect the dependency structure of the workflow. Hovering over the nodes in the DAG shows a tooltip for the app represented by the node and its task ID.

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](#)



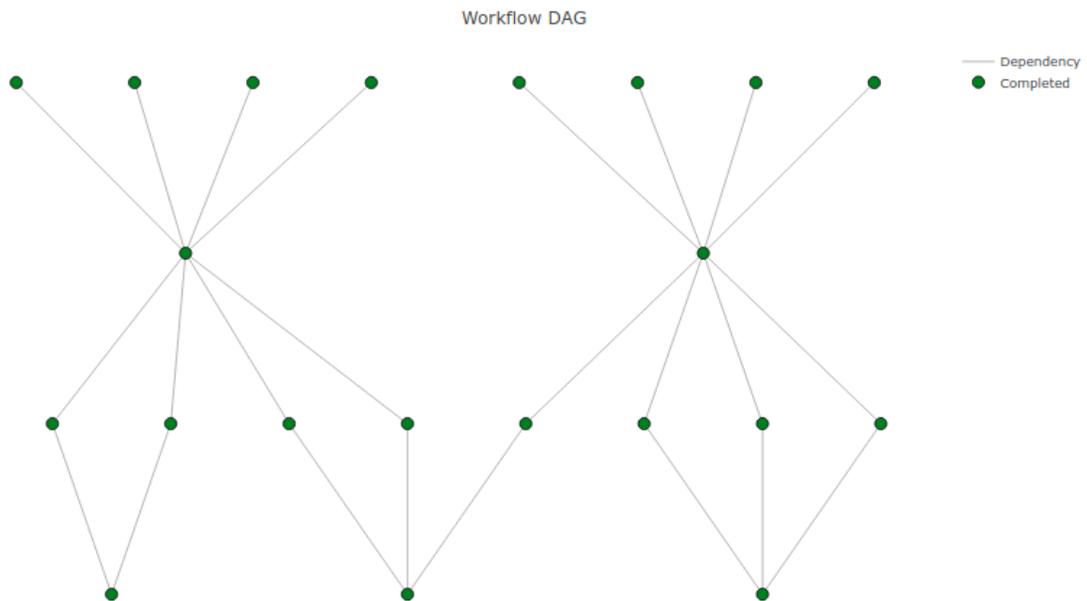
- Workflow DAG - with task states differentiated by colors: This visualization is useful to identify what tasks have been completed, failed, or are currently pending.

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](#)



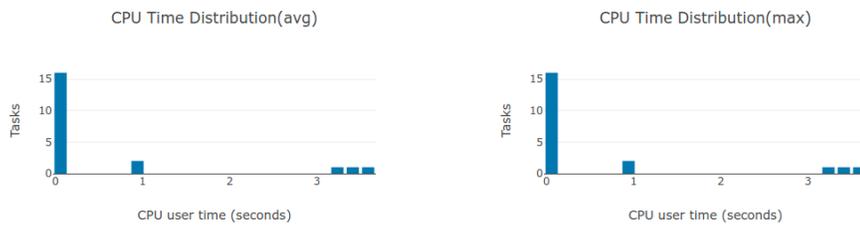
- **Workflow resource usage:** This visualization provides resource usage information at the workflow level. For example, cumulative CPU/Memory utilization across workers over time.

test_fan_inout.py

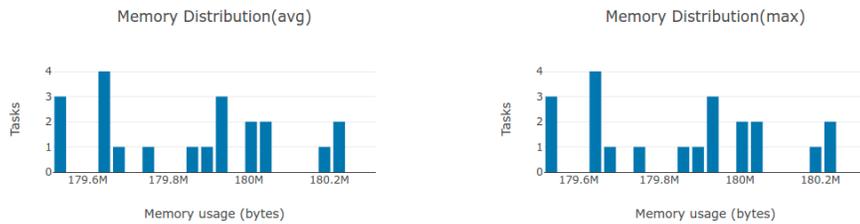
- **Started:** 2019-06-13 10:58:14
- **Completed:** 2019-06-13 10:58:39
- **Workflow duration:** 0:00:25
- **Owner:** yadu
- **host:** borgmachine2
- **rundir:** /home/yadu/src/parsl/parsl/tests/manual_tests/runinfo/000
- **tasks_failed_count:** 0
- **tasks_completed_count:** 21

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

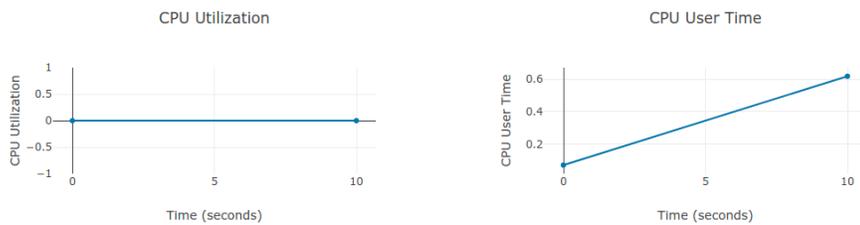
CPU Usage



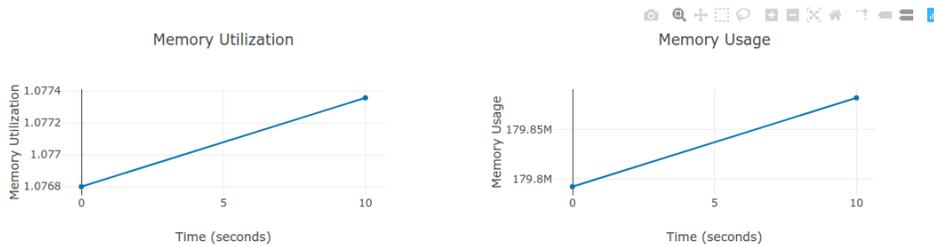
Memory Usage



CPU usage



Memory usage



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.

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()
    results.append(x)

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

```

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)

```

(continues on next page)

(continued from previous page)

```

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

message = generate(10)

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

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

```

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_z      (2 s)
doubled_z = wait_sleep_double(10, doubled_x, doubled_y)

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

```

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

The ParslPoolExecutor simplifies this pattern using the same interface as Python's native Executors.

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

# NOTE: Functions used by the ParslPoolExecutor do not use decorators
def generate(limit):
    """Generate a random integer and return it"""
    from random import randint
    return randint(1, limit)

with ParslPoolExecutor(config) as exec:
    outputs = pool.map(generate, range(1, 5))
```

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])
```

(continues on next page)

(continued from previous page)

```

@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())

```

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)

```

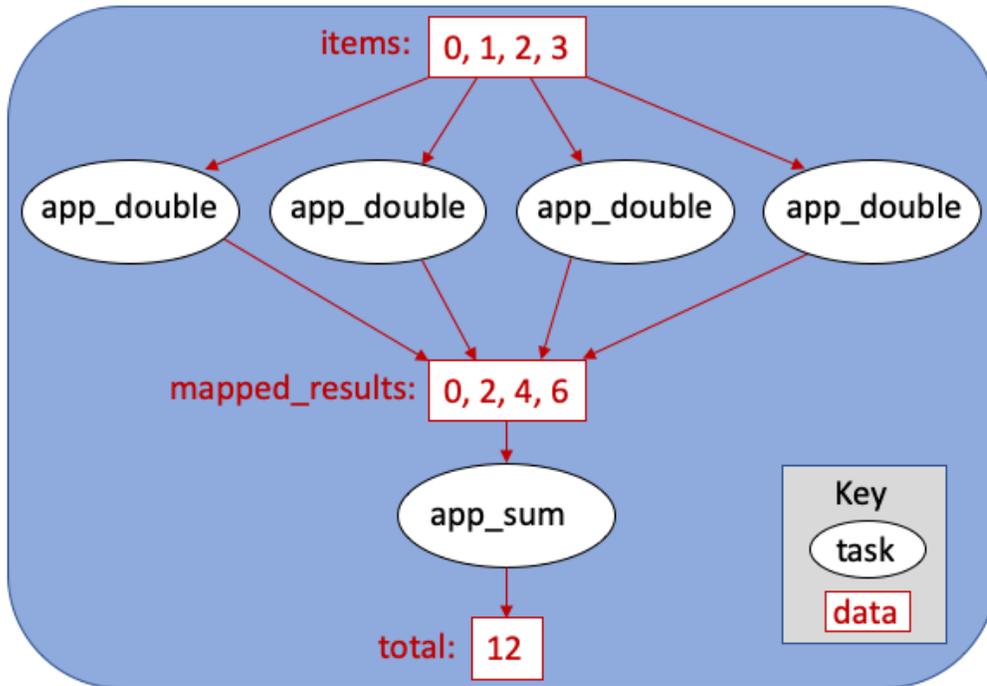
(continues on next page)

(continued from previous page)

```
# 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.



Caching expensive initialisation between tasks

Many tasks in workflows require a expensive “initialization” steps that, once performed, can be used across successive invocations for that task. For example, you may want to reuse a machine learning model for multiple interface tasks and avoid loading it onto GPUs more than once.

This [ExaWorks tutorial](#) gives examples of how to do this.

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
```

Lifted operators

Parsl allows some operators (`[]` and `.`) to be used on an `AppFuture` in a way that makes sense with those operators on the eventually returned result.

Lifted `[]` operator

When an app returns a complex structure such as a `dict` or a `list`, it is sometimes useful to pass an element of that structure to a subsequent task, without waiting for that subsequent task to complete.

To help with this, Parsl allows the `[]` operator to be used on an `AppFuture`. This operator will return another `AppFuture` that will complete after the initial future, with the result of `[]` on the value of the initial future.

The end result is that this assertion will hold:

```
fut = my_app()
assert fut['x'].result() == fut.result()[x]
```

but more concurrency will be available, as execution of the main workflow code will not stop to wait for `result()` to complete on the initial future.

`AppFuture` does not implement other methods commonly associated with dicts and lists, such as `len`, because those methods should return a specific type of result immediately, and that is not possible when the results are not available until the future.

If a key does not exist in the returned result, then the exception will appear in the Future returned by `[]`, rather than at the point that the `[]` operator is applied. This is because the valid values that can be used are not known until the underlying result is available.

Lifted `.` operator

The `.` operator works similarly to `[]` described above:

```
fut = my_app
assert fut.x.result() == fut.result().x
```

Attributes beginning with `_` are not lifted as this usually indicates an attribute that is used for internal purposes, and to try to avoid mixing protocols (such as iteration in for loops) defined on `AppFutures` vs protocols defined on the underlying result object.

Join Apps

Join apps, defined with the `@join_app` decorator, are a form of app that can launch other pieces of a workflow: for example a Parsl sub-workflow, or a task that runs in some other system.

Parsl sub-workflows

One reason for launching Parsl apps from inside a join app, rather than directly in the main workflow code, is because the definitions of those tasks are not known well enough at the start of the workflow.

For example, a workflow might run an expensive step to detect some objects in an image, and then on each object, run a further expensive step. Because the number of objects is not known at the start of the workflow, but instead only after an expensive step has completed, the subsequent tasks cannot be defined until after that step has completed.

In simple cases, the main workflow script can be stopped using `Future.result()` and join apps are not necessary, but in more complicated cases, that approach can severely limit concurrency.

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

Using Futures from other components

Sometimes, a workflow might need to incorporate tasks from other systems that run asynchronously but do not need a Parsl worker allocated for their entire run. An example of this is delegating some work into Globus Compute: work can be given to Globus Compute, but Parsl does not need to keep a worker allocated to that task while it runs. Instead, Parsl can be told to wait for the `Future` returned by Globus Compute to complete.

Usage

A `join_app` looks quite like a `python_app`, but should return one or more `Future` objects, rather than a value. Once the Python code has run, the app will wait for those Futures to complete without occupying a Parsl worker, and when those Futures complete, their contents will be the return value of the `join_app`.

For example:

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

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

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

Example of a Parsl sub-workflow

This example workflow shows a preprocessing step, followed by a middle stage that is chosen by the result of the pre-processing step (either option 1 or option 2) followed by a know post-processing step.

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

@python_app
def option_one(x):
    return x*2

@python_app
def option_two(x):
    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)

assert post_process(process(pre_process())) == "6"
```

- 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: even though process is invoked at the start of the workflow, it will execute later on, when the Future returned by `pre_process` has a value.

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

A Python app, if run in a `parsl.executors.ThreadPoolExecutor`, can launch more parsl apps; so a `python_app` implementation of `process()` would be able to inspect `x` and choose and invoke the appropriate `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, we want to invoke `post_process` at submission time near the start of workflow so that Parsl knows about as many tasks as possible. But we don't want it to execute until the value of the chosen `option_{one, two}` app is known.

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:
```

(continues on next page)

(continued from previous page)

```
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).

Example of invoking a Futures-driven task from another system

This example shows launching some activity in another system, without occupying a Parsl worker while that activity happens: in this example, work is delegated to Globus Compute, which performs the work elsewhere. When the work is completed, Globus Compute will put the result into the future that it returns, and then (because the Parsl app is a `@join_app`), that result will be used as the result of the Parsl app.

As above, the motivation for doing this inside an app, rather than in the top level is that sufficient information to launch the Globus Compute task might not be available at start of the workflow.

This workflow will run a first stage, `const_five`, on a Parsl worker, then using the result of that stage, pass the result as a parameter to a Globus Compute task, getting a `Future` from that submission. Then, the results of the Globus Compute task will be passed onto a second Parsl local task, `times_two`.

```
import parsl
from globus_compute_sdk import Executor

tutorial_endpoint_uuid = '4b116d3c-1703-4f8f-9f6f-39921e5864df'
gce = Executor(endpoint_id=tutorial_endpoint_uuid)

def increment_in_funcx(n):
    return n+1

@parsl.join_app
def increment_in_parsl(n):
    future = gce.submit(increment_in_funcx, n)
    return future

@parsl.python_app
def times_two(n):
    return n*2

@parsl.python_app
def const_five():
    return 5

parsl.load()

workflow = times_two(increment_in_parsl(const_five()))

r = workflow.result()

assert r == (5+1)*2
```

Terminology

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

Usage statistics collection

Parsl uses an **Opt-in** model to send 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.

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. (see *What is sent?* below).

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`*).

What is sent?

- IP address
- Run UUID
- Start and end times
- Number of executors used
- Number of failures
- Parsl and Python version
- OS and OS version

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.

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.

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?

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.

Plugins

Parsl has several places where code can be plugged in. Parsl usually provides several implementations that use each plugin point.

This page gives a brief summary of those places and why you might want to use them, with links to the API guide.

Executors

When the parsl dataflow kernel is ready for a task to run, it passes that task to an *ParslExecutor*. The executor is then responsible for running the task's Python code and returning the result. This is the abstraction that allows one executor to run code on the local submitting host, while another executor can run the same code on a large supercomputer.

Providers, Launchers and Channels

Some executors are based on blocks of workers (for example the *parsl.executors.HighThroughputExecutor*: the submit side requires a batch system (eg slurm, kubernetes) to start worker processes, which then execute tasks.

The particular way in which a system makes those workers start is implemented by providers and launchers.

An *ExecutionProvider* allows a command line to be submitted as a request to the underlying batch system to be run inside an allocation of nodes.

A *Launcher* modifies that command line when run inside the allocation to add on any wrappers that are needed to launch the command (eg srun inside slurm). Providers and launchers are usually paired together for a particular system type.

A *Channel* allows the commands used to interact with an *ExecutionProvider* to be executed on a remote system. The default channel executes commands on the local system, but a few variants of an `parsl.channels.SSHChannel` are provided.

File staging

Parsl can copy input files from an arbitrary URL into a task's working environment, and copy output files from a task's working environment to an arbitrary URL. A small set of data staging providers is installed by default, for `file://`, `http://` and `ftp://` URLs. More data staging providers can be added in the workflow configuration, in the `storage` parameter of the relevant *ParslExecutor*. Each provider should subclass the *Staging* class.

Default stdout/stderr name generation

Parsl can choose names for your bash apps stdout and stderr streams automatically, with the `parsl.AUTO_LOGNAME` parameter. The choice of path is made by a function which can be configured with the `std_autopath` parameter of Parsl *Config*. By default, `DataFlowKernel.default_std_autopath` will be used.

Memoization/checkpointing

When parsl memoizes/checkpoints an app parameter, it does so by computing a hash of that parameter that should be the same if that parameter is the same on subsequent invocations. This isn't straightforward to do for arbitrary objects, so parsl implements a checkpointing hash function for a few common types, and raises an exception on unknown types:

```
ValueError("unknown type for memoization ...")
```

You can plug in your own type-specific hash code for additional types that you need and understand using `id_for_memo`.

Invoking other asynchronous components

Parsl code can invoke other asynchronous components which return Futures, and integrate those Futures into the task graph: Parsl apps can be given any `concurrent.futures.Future` as a dependency, even if those futures do not come from invoking a Parsl app. This includes as the return value of a `join_app`.

An specific example of this is integrating Globus Compute tasks into a Parsl task graph. See *Example of invoking a Futures-driven task from another system*

Measuring performance with parsl-perf

parsl-perf is tool for making basic performance measurements of Parsl configurations.

It runs increasingly large numbers of no-op apps until a batch takes (by default) 120 seconds, giving a measurement of tasks per second.

This can give a basic measurement of some of the overheads in task execution.

parsl-perf must be invoked with a configuration file, which is a Python file containing a variable `config` which contains a `Config` object, or a function `fresh_config` which returns a `Config` object. The `fresh_config` format is the same as used with the pytest test suite.

To specify a `parsl_resource_specification` for tasks, add a `--resources` argument.

To change the target runtime from the default of 120 seconds, add a `--time` parameter.

For example:

```
$ python -m parsl.benchmark.perf --config parsl/tests/configs/workqueue_ex.py --
↳resources '{"cores":1, "memory":0, "disk":0}'
==== Iteration 1 ====
Will run 10 tasks to target 120 seconds runtime
Submitting tasks / invoking apps
warning: using plain-text when communicating with workers.
warning: use encryption with a key and cert when creating the manager.
All 10 tasks submitted ... waiting for completion
Submission took 0.008 seconds = 1248.676 tasks/second
Runtime: actual 3.668s vs target 120s
Tasks per second: 2.726

[...]

==== Iteration 4 ====
Will run 57640 tasks to target 120 seconds runtime
Submitting tasks / invoking apps
All 57640 tasks submitted ... waiting for completion
Submission took 34.839 seconds = 1654.487 tasks/second
Runtime: actual 364.387s vs target 120s
Tasks per second: 158.184
Cleaning up DFK
The end
```

FAQ

How can I debug a Parsl script?

Parsl interfaces with the Python logger and automatically logs Parsl-related messages a `runinfo` directory. The `runinfo` directory will be created in the folder from which you run the Parsl script and it will contain a series of subfolders for each time you run the code. Your latest run will be the largest number.

Alternatively, you can configure the file logger to write to an output file.

```
import parsl
```

(continues on next page)

(continued from previous page)

```
# 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.

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')
```

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.

Can I pass any Python object between apps?

This depends on the executor in use. The `parsl.executors.threads.ThreadPoolExecutor` can receive and return any Python object. Other executors will serialize their parameters and return values, so only objects which Parsl knows how to serialize can be passed.

Parsl knows how to serialize objects using the Pickle and Dill libraries.

Pickle provides a list of objects that it knows how to serialize: [What can be pickled and unpickled?](#).

Dill can serialize much more than Pickle, documented in the [dill documentation](#).

For objects that can't be pickled, use object specific methods to write the object into a file and use files to communicate between apps.

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 (...):
    ...
```

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 `parsl.executors.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 `pymq`. As a last resort, you can try: `pip install --upgrade --no-binary pymq pymq`, which forces re-compilation.

For the `parsl.executors.HighThroughputExecutor`, `address` is a keyword argument taken at initialization. Here is an example for the `parsl.executors.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
```

(continues on next page)

(continued from previous page)

```

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.

parsl.errors.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.errors.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](#).

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.

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'
```

You should usually install `parsl` using a package management tool such as `pip` or `conda`, ideally in a restricted environment such a `virtualenv` or a `conda` environment.

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...>
```

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.errors.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)
```

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
```

(continues on next page)

(continued from previous page)

```

config = Config(
    executors=[
        HighThroughputExecutor(
            label='ALCF_theta_local',
            provider=Cobalt(),
            worker_port_range=('50000', 55000),
            interchange_port_range=('50000', 55000)
        )
    ],
)

```

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](#).

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.

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 dill and pickle to serialize Python objects to/from functions. Therefore, Python apps can only use input and output objects that can be serialized by dill 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.

How do I cite Parsl?

To cite Parsl in publications, please use the following:

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

or

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

API Reference guide

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>	Decorator function for making join apps
<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>	

parsl.app.app.python_app

```
parsl.app.app.python_app(function: Callable | None = None, data_flow_kernel: DataFlowKernel | None =
    None, cache: bool = False, executors: List[str] | Literal['all'] = 'all',
    ignore_for_cache: Sequence[str] | None = None) → Callable
```

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.
- **ignore_for_cache** (*(sequence/None)*) – Names of arguments which will be ignored by the caching mechanism.

parsl.app.app.bash_app

```
parsl.app.app.bash_app(function: Callable | None = None, data_flow_kernel: DataFlowKernel | None = None,
    cache: bool = False, executors: List[str] | Literal['all'] = 'all', ignore_for_cache:
    Sequence[str] | None = None) → Callable
```

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.
- **ignore_for_cache** (*(list/None)*) – Names of arguments which will be ignored by the caching mechanism.

parsl.app.app.join_app

`parsl.app.app.join_app`(*function*: *Callable* | *None* = *None*, *data_flow_kernel*: *DataFlowKernel* | *None* = *None*, *cache*: *bool* = *False*, *ignore_for_cache*: *Sequence[str]* | *None* = *None*) → *Callable*

Decorator function for making join 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.
- **cache** (*bool*) – Enable caching of the app call. Default is False.
- **ignore_for_cache** (*(sequence/None)*) – Names of arguments which will be ignored by the caching mechanism.

parsl.dataflow.futures.AppFuture

`class parsl.dataflow.futures.AppFuture`(*task_record*: *TaskRecord*)

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_record*: *TaskRecord*) → *None*

Initialize the AppFuture.

Args:

KWargs:

- `task_record` : The `TaskRecord` for the task represented by this future.

Methods

<code>__init__(task_record)</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>	Return app stderr.
<code>stdout</code>	Return app stdout.
<code>tid</code>	

cancel() → bool

Cancel the future if possible.

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

cancelled() → bool

Return True if the future was cancelled.

property outputs: Sequence[DataFuture]

property stderr: None | str | DataFuture

Return app stderr. If stdout was specified as a string, then this property will return that string. If stdout was specified as a File, then this property will return a DataFuture representing that file stageout. TODO: this can be a tuple too I think?

property stdout: None | str | DataFuture

Return app stdout. If stdout was specified as a string, then this property will return that string. If stdout was specified as a File, then this property will return a DataFuture representing that file stageout. TODO: this can be a tuple too I think?

task_status() → str

Returns the status of the task that will provide the value for this future. This may not be in-sync with the result state of this future - for example, task_status might return ‘done’ but self.done() might not be true (which in turn means self.result() and self.exception() might block).

The actual status description strings returned by this method are likely to change over subsequent versions of parsl, as use-cases and infrastructure are worked out.

It is expected that the status values will be from a limited set of strings (so that it makes sense, for example, to group and count statuses from many futures).

It is expected that there might be a non-trivial cost in acquiring the status in future (for example, by communicating with a remote worker).

Returns: str

property tid: int

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__()

Methods

<code>__init__()</code>	
<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.

classmethod clear() → None

Clear the active DataFlowKernel so that a new one can be loaded.

classmethod dfk() → DataFlowKernel

Return the currently-loaded DataFlowKernel.

classmethod load(config: Config | None = None) → DataFlowKernel

Load a DataFlowKernel.

Parameters

config (-) – Configuration to load. This config will be passed to a new DataFlowKernel instantiation which will be set as the active DataFlowKernel.

Returns

The loaded DataFlowKernel object.

Return type

- DataFlowKernel

classmethod `wait_for_current_tasks()` → *None*

Waits for all tasks in the task list to be completed, by waiting for their AppFuture to be completed. This method will not necessarily wait for any tasks added after cleanup has started such as data stageout.

`parsl.monitoring.MonitoringHub`

```
class parsl.monitoring.MonitoringHub(hub_address: str, hub_port: int | None = None, hub_port_range: Tuple[int, int] = (55050, 56000), workflow_name: str | None = None, workflow_version: str | None = None, logging_endpoint: str | None = None, logdir: str | None = None, monitoring_debug: bool = False, resource_monitoring_enabled: bool = True, resource_monitoring_interval: float = 30)
```

```
__init__(hub_address: str, hub_port: int | None = None, hub_port_range: Tuple[int, int] = (55050, 56000), workflow_name: str | None = None, workflow_version: str | None = None, logging_endpoint: str | None = None, logdir: str | None = None, monitoring_debug: bool = False, resource_monitoring_enabled: bool = True, resource_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 UDP port to which workers will be able to deliver messages to the monitoring router. Note that despite the similar name, this is not related to `hub_port_range`. Default: `None`
- **hub_port_range** (*tuple(int, int)*) – The port range for a ZMQ channel from an executor process (for example, the interchange in the High Throughput Executor) to deliver monitoring messages to the monitoring router. Note that despite the similar name, this is not related to `hub_port`. Default: `(55050, 56000)`
- **workflow_name** (*str*) – The name for the workflow. Default to the name of the parsl script
- **workflow_version** (*str*) – The version of the workflow. Default to the beginning date-time of the parsl script
- **logging_endpoint** (*str*) – The database connection url for monitoring to log the information. These URLs follow RFC-1738, and can include username, password, hostname, database name. Default: `sqlite, in the configured run_dir`.
- **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 of information from the worker side. This will include environment information such as start time, hostname and block id, along with periodic 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. If set to `0`, only start and end information will be logged, and no periodic monitoring will be made. 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>send(mtype, message)</code>	
<code>start(run_id, dfk_run_dir, config_run_dir)</code>	

`close()` → None

`send(mtype: MessageType, message: Any)` → None

`start(run_id: str, dfk_run_dir: str, config_run_dir: str | PathLike)` → int

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 file 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.

parsl.config.Config

```
class parsl.config.Config(executors: Iterable[ParslExecutor] | None = None, app_cache: bool = True,
                          checkpoint_files: Sequence[str] | None = None, checkpoint_mode: None |
                          Literal['task_exit'] | Literal['periodic'] | Literal['dfk_exit'] | Literal['manual'] =
                          None, checkpoint_period: str | None = None, garbage_collect: bool = True,
                          internal_tasks_max_threads: int = 10, retries: int = 0, retry_handler:
                          Callable[[Exception, TaskRecord], float] | None = None, run_dir: str = 'runinfo',
                          std_autopath: Callable | None = None, strategy: str | None = 'simple',
                          strategy_period: float | int = 5, max_idletime: float = 120.0, monitoring:
                          MonitoringHub | None = None, usage_tracking: bool = False, initialize_logging:
                          bool = True)
```

Specification of Parsl configuration options.

Parameters

- **executors** (*sequence of ParslExecutor, optional*) – List (or other iterable) of *ParslExecutor* instances to use for executing tasks. Default is (*ThreadPoolExecutor()*).
- **app_cache** (*bool, optional*) – Enable app caching. Default is True.
- **checkpoint_files** (*sequence 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', 'periodic' or 'manual'. 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 (or available retry budget when using *retry_handler*) in case of failure. Default is 0.
- **retry_handler** (*function, optional*) – A user pluggable handler to decide if/how a task retry should happen. If no handler is specified, then each task failure incurs a retry cost of 1.
- **run_dir** (*str, optional*) – Path to run directory. Default is 'runinfo'.
- **std_autopath** (*function, optional*) – Sets the function used to generate stdout/stderr specifications when *parsl.AUTO_LOGPATH* is used. If no function is specified, generates paths that look like: *rundir/NNN/task_logs/X/task_{id}_{name}{label}.{out/err}*
- **strategy** (*str, optional*) – Strategy to use for scaling blocks according to workflow needs. Can be 'simple', 'htex_auto_scale', 'none' or *None*. If 'none' or *None*, dynamic scaling will be disabled. Default is 'simple'. The literal value *None* is deprecated.
- **strategy_period** (*float or int, optional*) – How often the scaling strategy should be executed. Default is 5 seconds.
- **max_idletime** (*float, optional*) – The maximum idle time allowed for an executor before strategy could shut down unused blocks. 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: Iterable[ParslExecutor] | None = None, app_cache: bool = True, checkpoint_files: Sequence[str] | None = None, checkpoint_mode: None | Literal['task_exit'] | Literal['periodic'] | Literal['dfk_exit'] | Literal['manual'] = None, checkpoint_period: str | None = None, garbage_collect: bool = True, internal_tasks_max_threads: int = 10, retries: int = 0, retry_handler: Callable[[Exception, TaskRecord], float] | None = None, run_dir: str = 'runinfo', std_autopath: Callable | None = None, strategy: str | None = 'simple', strategy_period: float | int = 5, max_idletime: float = 120.0, monitoring: MonitoringHub | None = None, usage_tracking: bool = False, initialize_logging: bool = True) → None
```

Methods

```
__init__([executors, app_cache, ...])
get_usage_information()
```

Attributes

```
executors
```

property **executors**: Sequence[ParslExecutor]

get_usage_information()

parsl.set_stream_logger

```
parsl.set_stream_logger(name: str = 'parsl', level: int = 10, format_string: str | None = None, stream: TextIOWrapper | None = None) → Logger
```

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.
- **stream** (-) – Specify sys.stdout or sys.stderr for stream. If not specified, the default stream for logging.StreamHandler is used.

Returns

- logger for specified name

parsl.set_file_logger

`parsl.set_file_logger(filename: str, name: str = 'parsl', level: int = 10, format_string: str | None = None) → Logger`

Add a file 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

- logger for specified name

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.

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.

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

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.

parsl.utils.get_all_checkpoints

`parsl.utils.get_all_checkpoints(rundir: str = 'runinfo')` → `Sequence[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`

parsl.utils.get_last_checkpoint

`parsl.utils.get_last_checkpoint(rundir: str = 'runinfo')` → `Sequence[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

Channels

<code>parsl.channels.base.Channel</code>	Channels are abstractions that enable Execution-Providers to talk to resource managers of remote compute facilities.
<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.

parsl.channels.base.Channel

class parsl.channels.base.Channel

Channels are abstractions that enable ExecutionProviders to talk to resource managers of remote compute facilities.

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 through SSH, or require two factor authentication.

The simplest Channel, *LocalChannel*, executes commands locally in a shell, while the *SSHChannel* authenticates you to remote systems.

Channels provide the ability to execute commands remotely, using the `execute_wait` method, and manipulate the remote file system using methods such as `push_file`, `pull_file` and `makedirs`.

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__()`

Methods

<code>__init__()</code>	
<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.
-------------------------	---------------------

abstract `abspath(path: str) → str`

Return the absolute path.

Parameters

path (*str*) – Path for which the absolute path will be returned.

abstract `close() → bool`

Closes the channel. Clean out any auth credentials.

Parameters

None –

Returns

Bool

abstract execute_wait(*cmd: str, walltime: int = 0, envs: Dict[str, str] = {}*) → Tuple[int, str, str]

Executes the cmd, with a defined walltime.

Parameters

- **cmd** (-) – Command string to execute over the channel
- **walltime** (-) – Timeout in seconds

KWargs:

- **envs** (Dict[str, str]) : Environment variables to push to the remote side

Returns

- (exit_code, stdout, stderr) (int, string, string)

abstract isdir(*path: str*) → bool

Return true if the path refers to an existing directory.

Parameters

path (*str*) – Path of directory to check.

abstract makedirs(*path: str, mode: int = 329, exist_ok: bool = False*) → None

Create a directory.

If intermediate directories do not exist, they will be created.

Parameters

- **path** (*str*) – Path of directory to create.
- **mode** (*int*) – Permissions (posix-style) for the newly-created directory.
- **exist_ok** (*bool*) – If False, raise an OSError if the target directory already exists.

abstract pull_file(*remote_source: str, local_dir: str*) → str

Transport file on the remote side to a local directory

Parameters

- **remote_source** (*string*) – remote_source
- **local_dir** (*string*) – Local directory to copy to

Returns

destination_path (string)

abstract push_file(*source: str, dest_dir: str*) → str

Channel will take care of moving the file from source to the destination directory

Parameters

- **source** (*string*) – Full filepath of the file to be moved
- **dest_dir** (*string*) – Absolute path of the directory to move to

Returns

destination_path (string)

abstract property script_dir: str

This is a property. Returns the directory assigned for storing all internal scripts such as scheduler submit scripts. This is usually where error logs from the scheduler would reside on the channel destination side.

Parameters**None** (-) –**Returns**

- Channel script dir

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__</code> (<i>userhome</i> , <i>envs</i> , <i>script_dir</i>)	Initialize the local channel.
<code>abspath</code> (<i>path</i>)	Return the absolute path.
<code>close</code> ()	There's nothing to close here, and this really doesn't do anything
<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.
<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>source</i> , <i>dest_dir</i>)	If the source files dirpath is the same as <i>dest_dir</i> , a copy is not necessary, and nothing is done.

Attributes

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

abspath(*path*)

Return the absolute path.

Parameters

path (*str*) – Path for which the absolute path will be returned.

close()

There's nothing to close here, and this really doesn't do anything

Returns

- False, because it really did not “close” this channel.

execute_wait(*cmd*, *walltime=None*, *envs={}*)

Synchronously execute a commandline string on the shell.

Parameters

- **cmd** (-) – Commandline string to execute
- **walltime** (-) – walltime in seconds, this is not really used now.

Kwargs:

- **envs** (dict) : Dictionary of env variables. This will be used to override the envs set at channel initialization.

Returns

Return code from the execution, -1 on fail - **stdout** : stdout string - **stderr** : stderr string

Return type

- **retcode**

Raises: None.

isdir(*path*)

Return true if the path refers to an existing directory.

Parameters

path (*str*) – Path of directory to check.

makedirs(*path*, *mode=448*, *exist_ok=False*)

Create a directory.

If intermediate directories do not exist, they will be created.

Parameters

- **path** (*str*) – Path of directory to create.
- **mode** (*int*) – Permissions (posix-style) for the newly-created directory.
- **exist_ok** (*bool*) – If False, raise an OSError if the target directory already exists.

pull_file(*remote_source*, *local_dir*)

Transport file on the remote side to a local directory

Parameters

- **remote_source** (*string*) – remote_source
- **local_dir** (*string*) – Local directory to copy to

Returns

destination_path (string)

push_file(*source*, *dest_dir*)

If the source files dirpath is the same as dest_dir, a copy is not necessary, and nothing is done. Else a copy is made.

Parameters

- **source** (-) – Path to the source file
- **dest_dir** (-) – Path to the directory to which the files is to be copied

Returns

Absolute path of the destination file

Return type

- `destination_path` (String)

Raises

- **FileCopyException** – If file copy failed.

property `script_dir`

This is a property. Returns the directory assigned for storing all internal scripts such as scheduler submit scripts. This is usually where error logs from the scheduler would reside on the channel destination side.

Parameters

None (-) –

Returns

- Channel script dir

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__(hostname[, username, password, ...])</code>	Initialize a persistent connection to the remote system.
<code>abspath(path)</code>	Return the absolute path on the remote side.
<code>close()</code>	Closes the channel.
<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 on the remote side.
<code>prepend_envs(cmd[, env])</code>	
<code>pull_file(remote_source, local_dir)</code>	Transport file on the remote side to a local directory
<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.
-------------------------	---------------------

`abspath(path)`

Return the absolute path on the remote side.

Parameters

path (*str*) – Path for which the absolute path will be returned.

`close()`

Closes the channel. Clean out any auth credentials.

Parameters

None –

Returns

Bool

`execute_wait(cmd, walltime=2, envs={})`

Synchronously execute a commandline string on the shell.

Parameters

- **cmd** (-) – Commandline string to execute
- **walltime** (-) – walltime in seconds

Kwargs:

- **envs** (dict) : Dictionary of env variables

Returns

Return code from the execution, -1 on fail - stdout : stdout string - stderr : stderr string

Return type

- retcode

Raises: None.

isdir(*path*)

Return true if the path refers to an existing directory.

Parameters

path (*str*) – Path of directory on the remote side to check.

makedirs(*path, mode=448, exist_ok=False*)

Create a directory on the remote side.

If intermediate directories do not exist, they will be created.

Parameters

- **path** (*str*) – Path of directory on the remote side to create.
- **mode** (*int*) – Permissions (posix-style) for the newly-created directory.
- **exist_ok** (*bool*) – If False, raise an OSError if the target directory already exists.

prepend_envs(*cmd, env={}*)

pull_file(*remote_source, local_dir*)

Transport file on the remote side to a local directory

Parameters

- **remote_source** (-) – remote_source
- **local_dir** (-) – Local directory to copy to

Returns

Local path to file

Return type

- str

Raises

- - **FileExists** – Name collision at local directory.
- - **FileCopyException** – FileCopy failed.

push_file(*local_source, remote_dir*)

Transport a local file to a directory on a remote machine

Parameters

- **local_source** (-) – Path
- **remote_dir** (-) – Remote path

Returns

Path to copied file on remote machine

Return type

- str

Raises

- - **BadScriptPath** – if script path on the remote side is bad
- - **BadPermsScriptPath** – You do not have perms to make the channel script dir
- - **FileCopyException** – FileCopy failed.

property script_dir

This is a property. Returns the directory assigned for storing all internal scripts such as scheduler submit scripts. This is usually where error logs from the scheduler would reside on the channel destination side.

Parameters

None (-) –

Returns

- Channel script dir

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__(hostname[, username, script_dir, ...])</code>	Initialize a persistent connection to the remote system.
<code>abspath(path)</code>	Return the absolute path on the remote side.
<code>close()</code>	Closes the channel.
<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 on the remote side.
<code>prepend_envs(cmd[, env])</code>	
<code>pull_file(remote_source, local_dir)</code>	Transport file on the remote side to a local directory
<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.
-------------------------	---------------------

`close()`

Closes the channel. Clean out any auth credentials.

Parameters

None –

Returns

Bool

`execute_wait(cmd, walltime=60, envs={})`

Synchronously execute a commandline string on the shell.

This command does *NOT* honor walltime currently.

Parameters

cmd (-) – Commandline string to execute

Kwargs:

- `walltime` (int) : walltime in seconds
- `envs` (dict) : Dictionary of env variables

Returns

Return code from the execution, -1 on fail - `stdout` : stdout string - `stderr` : stderr string

Return type

- `retcode`

Raises: None.

`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__(hostname[, username, password, ...])</code>	Initialize a persistent connection to the remote system.
<code>abspath(path)</code>	Return the absolute path on the remote side.
<code>close()</code>	Closes the channel.
<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 on the remote side.
<code>prepend_envs(cmd[, env])</code>	
<code>pull_file(remote_source, local_dir)</code>	Transport file on the remote side to a local directory
<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.
-------------------------	---------------------

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.

parsl.app.futures.DataFuture

class `parsl.app.futures.DataFuture`(*fut: Future, file_obj: File, tid: int | None = 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: Future, file_obj: File, tid: int | None = None*) → 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.

cancel()

Cancel the future if possible.

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

cancelled()

Return True if the future was cancelled.

property filename

Filepath of the File object this datafuture represents.

property filepath

Filepath of the File object this datafuture represents.

parent_callback(*parent_fu*)

Callback from executor future to update the parent.

Updates the future with the result (the File object) or the parent future's exception.

Parameters

parent_fu (-) – Future returned by the executor along with callback

Returns

- None

running()

Return True if the future is currently executing.

property tid

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

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.
<code>stage_in(file, input, executor)</code>	Transport the input from the input source to the executor, if it is file-like, returning a DataFuture 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.

optionally_stage_in(*input*, *func*, *executor*)

replace_task(*file*: File, *func*: Callable, *executor*: str) → Callable

This will give staging providers the chance to wrap (or replace entirely!) the task function.

replace_task_stage_out(*file*: File, *func*: Callable, *executor*: str) → Callable

This will give staging providers the chance to wrap (or replace entirely!) the task function.

stage_in(*file*: File, *input*: Any, *executor*: str) → Any

Transport the input from the input source to the executor, if it is file-like, returning a DataFuture that wraps the stage-in operation.

If no staging in is required - because the `file` parameter is not file-like, then return that parameter unaltered.

Parameters

- **self** (-) –
- **input** (-) – input to stage in. If this is a File or a DataFuture, stage in tasks will be launched with appropriate dependencies. Otherwise, no stage-in will be performed.
- **executor** (-) – an executor the file is going to be staged in to.

stage_out(*file*: File, *executor*: str, *app_fu*: Future) → Future | None

Transport the file from the local filesystem to the remote Globus endpoint.

This function returns either a Future which should complete when the stageout is complete, or None, if no staging needs to be waited for.

Parameters

- **self** (-) –
- **file** (-) –
- **executor** (-) –
- **app_fu** (-) – complete before stageout begins.

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__()`

Methods

<code>__init__()</code>	
<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.

can_stage_in(*file*: File) → bool

Given a File object, decide if this staging provider can stage the file. Usually this is based on the URL scheme, but does not have to be. If this returns True, then other methods of this Staging object will be called to perform the staging.

can_stage_out(*file*: File) → bool

Like `can_stage_in`, but for staging out.

replace_task(*dm*: DataManager, *executor*: str, *file*: File, *func*: Callable) → Callable | None

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*: *DataManager*, *executor*: *str*, *file*: *File*, *func*: *Callable*) → *Callable* | *None*

For a file to be staged out, optionally return a replacement app function, which usually should be the original app function wrapped in staging code.

stage_in(*dm*: *DataManager*, *executor*: *str*, *file*: *File*, *parent_fut*: *Future* | *None*) → *DataFuture* | *None*

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.

This call will be made with a fresh copy of the *File* that may be modified for the purposes of this particular staging operation, rather than the original application-provided *File*. This allows staging specific information (primarily localpath) to be set on the *File* without interfering with other stagings of the same *File*.

The call can return a:

- *DataFuture*: the corresponding task input parameter will be replaced by the *DataFuture*, and the main task will not run until that *DataFuture* is complete. The *DataFuture* result should be the file object as passed in.
- *None*: the corresponding task input parameter will be replaced by a suitable automatically generated replacement that contains the *File* fresh copy, or is the fresh copy.

stage_out(*dm*: *DataManager*, *executor*: *str*, *file*: *File*, *app_fu*: *Future*) → *Future* | *None*

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.

Even though it should set up stageout, it will be invoked before the task executes. Any work which needs to happen after the main task should depend on *app_fu*.

For a given file, either return a *Future* which completes when stageout is complete, or return *None* to indicate that no stageout action need be waited for. When that *Future* completes, parsl will mark the relevant output *DataFuture* complete.

Note the asymmetry here between *stage_in* and *stage_out*: this can return any *Future*, while *stage_in* must return a *DataFuture*.

parsl.data_provider.files.File

class `parsl.data_provider.files.File`(*url*: *PathLike* | *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*: *PathLike* | *str*)

Construct a *File* object from a url string.

Parameters

url (-) – url of the file e.g. - 'input.txt' - `pathlib.Path('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

<code>__init__(url)</code>	Construct a File object from a url string.
<code>cleancopy()</code>	Returns a copy of the file containing only the global immutable state, without any mutable site-local local_path information.

Attributes

<code>filepath</code>	Return the resolved filepath on the side where it is called from.
-----------------------	---

`cleancopy()` → *File*

Returns a copy of the file containing only the global immutable state, without any mutable site-local local_path information. The returned File object will be as the original object was when it was constructed.

property filepath: `str`

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

The appropriate filepath will be returned when called from within an app running remotely as well as regular python on the submit side.

Only file: scheme URLs make sense to have a submit-side path, as other URLs are not accessible through POSIX file access.

Returns

- filepath

`parsl.data_provider.ftp.FTPSeparateTaskStaging`

class `parsl.data_provider.ftp.FTPSeparateTaskStaging`

Performs FTP staging as a separate parsl level task.

`__init__()`

Methods

<code>__init__()</code>	
<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.

`can_stage_in(file)`

Given a File object, decide if this staging provider can stage the file. Usually this is based on the URL scheme, but does not have to be. If this returns True, then other methods of this Staging object will be called to perform the staging.

`stage_in(dm, executor, file, parent_fut)`

This call gives the staging provider an opportunity to prepare for stage-in and to launch arbitrary tasks which must complete as part of stage-in.

This call will be made with a fresh copy of the File that may be modified for the purposes of this particular staging operation, rather than the original application-provided File. This allows staging specific information (primarily localpath) to be set on the File without interfering with other stagings of the same File.

The call can return a:

- DataFuture: the corresponding task input parameter will be replaced by the DataFuture, and the main task will not run until that DataFuture is complete. The DataFuture result should be the file object as passed in.
- None: the corresponding task input parameter will be replaced by a suitable automatically generated replacement that contains the File fresh copy, or is the fresh copy.

`parsl.data_provider.ftp.FTPInTaskStaging`

`class parsl.data_provider.ftp.FTPInTaskStaging`

Performs FTP staging as a wrapper around the application task.

`__init__()`

Methods

<code>__init__()</code>	
<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.

`can_stage_in(file)`

Given a File object, decide if this staging provider can stage the file. Usually this is based on the URL scheme, but does not have to be. If this returns True, then other methods of this Staging object will be called to perform the staging.

`replace_task(dm, executor, file, f)`

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.

`stage_in(dm, executor, file, parent_fut)`

This call gives the staging provider an opportunity to prepare for stage-in and to launch arbitrary tasks which must complete as part of stage-in.

This call will be made with a fresh copy of the File that may be modified for the purposes of this particular staging operation, rather than the original application-provided File. This allows staging specific information (primarily localpath) to be set on the File without interfering with other stagings of the same File.

The call can return a:

- DataFuture: the corresponding task input parameter will be replaced by the DataFuture, and the main task will not run until that DataFuture is complete. The DataFuture result should be the file object as passed in.
- None: the corresponding task input parameter will be replaced by a suitable automatically generated replacement that contains the File fresh copy, or is the fresh copy.

parsl.data_provider.file_noop.NoOpFileStaging

```
class parsl.data_provider.file_noop.NoOpFileStaging
```

```
    __init__()
```

Methods

<code>__init__()</code>	
<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.

`can_stage_in(file)`

Given a File object, decide if this staging provider can stage the file. Usually this is based on the URL scheme, but does not have to be. If this returns True, then other methods of this Staging object will be called to perform the staging.

`can_stage_out(file)`

Like `can_stage_in`, but for staging out.

parsl.data_provider.globus.GlobusStaging

```
class parsl.data_provider.globus.GlobusStaging(endpoint_uuid: str, endpoint_path: str | None = None,
                                              local_path: str | None = 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: str | None = None, local_path: str | None = None)
```

Methods

<code>__init__(endpoint_uuid[, endpoint_path, ...])</code>	
<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.
<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.

`can_stage_in(file)`

Given a File object, decide if this staging provider can stage the file. Usually this is based on the URL scheme, but does not have to be. If this returns True, then other methods of this Staging object will be called to perform the staging.

`can_stage_out(file)`

Like `can_stage_in`, but for staging out.

`initialize_globus()`

`stage_in(dm, executor, file, parent_fut)`

This call gives the staging provider an opportunity to prepare for stage-in and to launch arbitrary tasks which must complete as part of stage-in.

This call will be made with a fresh copy of the File that may be modified for the purposes of this particular staging operation, rather than the original application-provided File. This allows staging specific information (primarily localpath) to be set on the File without interfering with other stagings of the same File.

The call can return a:

- DataFuture: the corresponding task input parameter will be replaced by the DataFuture, and the main task will not run until that DataFuture is complete. The DataFuture result should be the file object as passed in.
- None: the corresponding task input parameter will be replaced by a suitable automatically generated replacement that contains the File fresh copy, or is the fresh copy.

`stage_out(dm, executor, file, app_fu)`

This call gives the staging provider an opportunity to prepare for stage-out and to launch arbitrary tasks which must complete as part of stage-out.

Even though it should set up stageout, it will be invoked before the task executes. Any work which needs to happen after the main task should depend on `app_fu`.

For a given file, either return a Future which completes when stageout is complete, or return None to indicate that no stageout action need be waited for. When that Future completes, parsl will mark the relevant output DataFuture complete.

Note the asymmetry here between stage_in and stage_out: this can return any Future, while stage_in must return a DataFuture.

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__()`

Methods

<code>__init__()</code>	
<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.

`can_stage_in(file)`

Given a File object, decide if this staging provider can stage the file. Usually this is based on the URL scheme, but does not have to be. If this returns True, then other methods of this Staging object will be called to perform the staging.

`stage_in(dm, executor, file, parent_fut)`

This call gives the staging provider an opportunity to prepare for stage-in and to launch arbitrary tasks which must complete as part of stage-in.

This call will be made with a fresh copy of the File that may be modified for the purposes of this particular staging operation, rather than the original application-provided File. This allows staging specific information (primarily localpath) to be set on the File without interfering with other stagings of the same File.

The call can return a:

- DataFuture: the corresponding task input parameter will be replaced by the DataFuture, and the main task will not run until that DataFuture is complete. The DataFuture result should be the file object as passed in.
- None: the corresponding task input parameter will be replaced by a suitable automatically generated replacement that contains the File fresh copy, or is the fresh copy.

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__()`

Methods

<code>__init__()</code>	
<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.

`can_stage_in(file)`

Given a File object, decide if this staging provider can stage the file. Usually this is based on the URL scheme, but does not have to be. If this returns True, then other methods of this Staging object will be called to perform the staging.

`replace_task(dm, executor, file, f)`

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.

`stage_in(dm, executor, file, parent_fut)`

This call gives the staging provider an opportunity to prepare for stage-in and to launch arbitrary tasks which must complete as part of stage-in.

This call will be made with a fresh copy of the File that may be modified for the purposes of this particular staging operation, rather than the original application-provided File. This allows staging specific information (primarily localpath) to be set on the File without interfering with other stagings of the same File.

The call can return a:

- DataFuture: the corresponding task input parameter will be replaced by the DataFuture, and the main task will not run until that DataFuture is complete. The DataFuture result should be the file object as passed in.
- None: the corresponding task input parameter will be replaced by a suitable automatically generated replacement that contains the File fresh copy, or is the fresh copy.

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)`

Methods

<code>__init__(hostname)</code>	
<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.

can_stage_in(file)

Given a File object, decide if this staging provider can stage the file. Usually this is based on the URL scheme, but does not have to be. If this returns True, then other methods of this Staging object will be called to perform the staging.

can_stage_out(file)

Like `can_stage_in`, but for staging out.

replace_task(dm, executor, file, f)

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, f*)

For a file to be staged out, optionally return a replacement app function, which usually should be the original app function wrapped in staging code.

stage_in(*dm, executor, file, parent_fut*)

This call gives the staging provider an opportunity to prepare for stage-in and to launch arbitrary tasks which must complete as part of stage-in.

This call will be made with a fresh copy of the File that may be modified for the purposes of this particular staging operation, rather than the original application-provided File. This allows staging specific information (primarily localpath) to be set on the File without interfering with other stagings of the same File.

The call can return a:

- DataFuture: the corresponding task input parameter will be replaced by the DataFuture, and the main task will not run until that DataFuture is complete. The DataFuture result should be the file object as passed in.
- None: the corresponding task input parameter will be replaced by a suitable automatically generated replacement that contains the File fresh copy, or is the fresh copy.

stage_out(*dm, executor, file, parent_fut*)

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.

Even though it should set up stageout, it will be invoked before the task executes. Any work which needs to happen after the main task should depend on app_fu.

For a given file, either return a Future which completes when stageout is complete, or return None to indicate that no stageout action need be waited for. When that Future completes, parsl will mark the relevant output DataFuture complete.

Note the asymmetry here between stage_in and stage_out: this can return any Future, while stage_in must return a DataFuture.

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.status_handling.BlockProviderExecutor</code>	A base class for executors which scale using blocks.
<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.taskvine.TaskVineExecutor</code>	Executor to use TaskVine dynamic workflow system
<code>parsl.executors.FluxExecutor</code>	Executor that uses Flux to schedule and run jobs.
<code>parsl.executors.radical.RadicalPilotExecutor</code>	Executor is designed for executing heterogeneous tasks

parsl.executors.base.ParslExecutor

```
class parsl.executors.base.ParslExecutor(*, hub_address: str | None = None, hub_port: int | None =
None, monitoring_radio: MonitoringRadio | None = None,
run_dir: str = '.', run_id: str | None = None)
```

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

This is an abstract base class that only enforces concrete implementations of functionality by the child classes.

Can be used as a context manager. On exit, calls `self.shutdown()` with no arguments and re-raises any thrown exception.

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.

Per-executor monitoring behaviour can be influenced by exposing:

radio_mode: str - a string describing which radio mode should be used to
send task resource data back to the submit side.

An executor may optionally expose:

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

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

```
__init__(*, hub_address: str | None = None, hub_port: int | None = None, monitoring_radio:
MonitoringRadio | None = None, run_dir: str = '.', run_id: str | None = None)
```

Methods

<code>__init__(*[, hub_address, hub_port, ...])</code>	
<code>monitor_resources()</code>	Should resource monitoring happen for tasks on running on this executor?
<code>shutdown()</code>	Shutdown the executor.
<code>start()</code>	Start the executor.
<code>submit(func, resource_specification, *args, ...)</code>	Submit.

Attributes

<code>hub_address</code>	Address to the Hub for monitoring.
<code>hub_port</code>	Port to the Hub for monitoring.
<code>label</code>	
<code>monitoring_radio</code>	Local radio for sending monitoring messages
<code>radio_mode</code>	
<code>run_dir</code>	Path to the run directory.
<code>run_id</code>	UUID for the enclosing DFK.

property `hub_address`: `str` | `None`

Address to the Hub for monitoring.

property `hub_port`: `int` | `None`

Port to the Hub for monitoring.

label: `str` = `'undefined'`

monitor_resources() → `bool`

Should resource monitoring happen for tasks on running on this executor?

Parsl resource monitoring conflicts with execution styles which use threads, and can deadlock while running.

This function allows resource monitoring to be disabled per executor implementation.

property `monitoring_radio`: `MonitoringRadio` | `None`

Local radio for sending monitoring messages

radio_mode: `str` = `'udp'`

property `run_dir`: `str`

Path to the run directory.

property `run_id`: `str` | `None`

UUID for the enclosing DFK.

abstract `shutdown`() → `None`

Shutdown the executor.

This includes all attached resources such as workers and controllers.

abstract `start`() → `None`

Start the executor.

Any spin-up operations (for example: starting thread pools) should be performed here.

abstract `submit`(*func*: `Callable`, *resource_specification*: `Dict[str, Any]`, **args*: `Any`, ***kwargs*: `Any`) → `Future`

Submit.

The executor can optionally set a `parsl_executor_task_id` attribute on the `Future` that it returns, and in that case, `parsl` will log a relationship between the executor's task ID and `parsl` level `try/task` IDs.

parsl.executors.status_handling.BlockProviderExecutor

```
class parsl.executors.status_handling.BlockProviderExecutor(* , provider: ExecutionProvider |
    None, block_error_handler: bool |
    Callable[[BlockProviderExecutor,
    Dict[str, JobStatus]], None])
```

A base class for executors which scale using blocks.

This base class is intended to help with executors which:

- use blocks of workers to execute tasks
- blocks of workers are launched on a batch system through an *ExecutionProvider*

An implementing class should implement the abstract methods required by *ParslExecutor* to submit tasks, as well as *BlockProviderExecutor* abstract methods to provide the executor-specific command to start a block of workers (the `_get_launch_command` method), and some basic scaling information (`outstanding` and `workers_per_node` properties).

This base class provides a `scale_out` method which will launch new blocks. It does not provide a `scale_in` method, because scale-in behaviour is not well defined in the Parsl scaling model and so behaviour is left to individual executors.

Parsl scaling will provide scaling between `min_blocks` and `max_blocks` by invoking `scale_out`, but it will not initialize the blocks requested by any `init_blocks` parameter. Subclasses must implement that behaviour themselves.

BENC: TODO: block error handling: maybe I want this more user pluggable? I'm not sure of use cases for switchability at the moment beyond "yes or no"

```
__init__(* , provider: ExecutionProvider | None, block_error_handler: bool |
    Callable[[BlockProviderExecutor, Dict[str, JobStatus]], None])
```

Methods

<code>__init__(*, provider, block_error_handler)</code>	
<code>create_monitoring_info(status)</code>	Create a monitoring message for each block based on the poll status.
<code>handle_errors(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>poll_facade()</code>	
<code>scale_in(blocks)</code>	Scale in method.
<code>scale_in_facade(n[, max_idletime])</code>	
<code>scale_out([blocks])</code>	Scales out the number of blocks by "blocks"
<code>scale_out_facade(n)</code>	
<code>send_monitoring_info(status)</code>	
<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>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>label</code>	
<code>monitoring_radio</code>	Local radio for sending monitoring messages
<code>outstanding</code>	This should return the number of tasks that the executor has been given to run (waiting to run, and running now)
<code>provider</code>	
<code>radio_mode</code>	
<code>run_dir</code>	Path to the run directory.
<code>run_id</code>	UUID for the enclosing DFK.
<code>status_facade</code>	Return the status of all jobs/blocks of the executor of this poller.
<code>status_polling_interval</code>	Returns the interval, in seconds, at which the status method should be called.
<code>tasks</code>	
<code>workers_per_node</code>	

property `bad_state_is_set`

Returns true if this executor is in an irrecoverable error state. If this method returns true, `property:executor_exception` should contain an exception indicating the cause.

create_monitoring_info(*status: Dict[str, JobStatus]*) → Sequence[object]

Create a monitoring message for each block based on the poll status.

property `executor_exception`

Returns an exception that indicates why this executor is in an irrecoverable state.

handle_errors(*status: Dict[str, JobStatus]*) → None

This method is called by the error management infrastructure after a status poll. The executor implementing this method is then responsible for detecting abnormal conditions based on the status of submitted jobs. If the executor does not implement any special error handling, this method should return False, in which case a generic error handling scheme will be used. `:param status:` status of all jobs launched by this executor

abstract property `outstanding: int`

This should return the number of tasks that the executor has been given to run (waiting to run, and running now)

poll_facade() → None

property `provider`

abstract scale_in(*blocks: int*) → List[str]

Scale in method.

Cause the executor to reduce the number of blocks by count.

Returns

A list of block ids corresponding to the blocks that were removed.

scale_in_facade(*n: int, max_idletime: float | None = None*) → List[str]

scale_out(*blocks: int = 1*) → List[str]

Scales out the number of blocks by “blocks”

scale_out_facade(*n: int*) → List[str]

send_monitoring_info(*status: Dict*) → None

set_bad_state_and_fail_all(*exception: Exception*)

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. The executor is responsible for checking :method:bad_state_is_set() in the :method:submit() method and raising the appropriate exception, which is available through :method:executor_exception().

status() → Dict[str, JobStatus]

Return the status of all jobs/blocks currently known to this executor.

Returns

a dictionary mapping block ids (in string) to job status

property status_facade: Dict[str, JobStatus]

Return the status of all jobs/blocks of the executor of this poller.

Returns

a dictionary mapping block ids (in string) to job status

property status_polling_interval

Returns the interval, in seconds, at which the status method should be called. The assumption here is that, once initialized, an executor’s polling interval is fixed. In practice, at least given the current situation, the executor uses a single task provider and this method is a delegate to the corresponding method in the provider.

Returns

the number of seconds to wait between calls to status() or zero if no polling should be done

property tasks: Dict[object, Future]

abstract property workers_per_node: int | float

parsl.executors.ThreadPoolExecutor

```
class parsl.executors.ThreadPoolExecutor(label: str = 'threads', max_threads: int | None = 2,
                                         thread_name_prefix: str = "", storage_access: List[Staging] |
                                         None = None, working_dir: str | None = None)
```

A thread-based executor.

Parameters

- **max_threads** (*Optional[int]*) – Number of threads. Default is 2.

- **thread_name_prefix** (*string*) – Thread name prefix
- **storage_access** (list of *Staging*) – Specifications for accessing data this executor remotely.

`__init__`(*label: str = 'threads', max_threads: int | None = 2, thread_name_prefix: str = "", storage_access: List[Staging] | None = None, working_dir: str | None = None*)

Methods

<code>__init__</code> ([<i>label</i> , <i>max_threads</i> , ...])	
<code>monitor_resources</code> ()	Resource monitoring sometimes deadlocks when using threads, so this function returns false to disable it.
<code>shutdown</code> ([<i>block</i>])	Shutdown the ThreadPool.
<code>start</code> ()	Start the executor.
<code>submit</code> (<i>func</i> , <i>resource_specification</i> , * <i>args</i> , ...)	Submits work to the thread pool.

Attributes

<code>hub_address</code>	Address to the Hub for monitoring.
<code>hub_port</code>	Port to the Hub for monitoring.
<code>label</code>	
<code>monitoring_radio</code>	Local radio for sending monitoring messages
<code>radio_mode</code>	
<code>run_dir</code>	Path to the run directory.
<code>run_id</code>	UUID for the enclosing DFK.

`monitor_resources()`

Resource monitoring sometimes deadlocks when using threads, so this function returns false to disable it.

`shutdown(block=True)`

Shutdown the ThreadPool. The underlying concurrent.futures thread pool implementation will not terminate tasks that are being executed, because it does not provide a mechanism to do that. With `block` set to false, this will return immediately and it will appear as if the DFK is shut down, but the python process will not be able to exit until the thread pool has emptied out by task completions. In either case, this can be a very long wait.

Kwargs:

- `block` (Bool): To block for confirmations or not

`start()`

Start the executor.

Any spin-up operations (for example: starting thread pools) should be performed here.

`submit(func, resource_specification, *args, **kwargs)`

Submits work to the thread pool.

This method is simply pass through and behaves like a submit call as described here [Python docs](#):

parsl.executors.HighThroughputExecutor

class parsl.executors.HighThroughputExecutor(*label: str = 'HighThroughputExecutor', provider:*

```

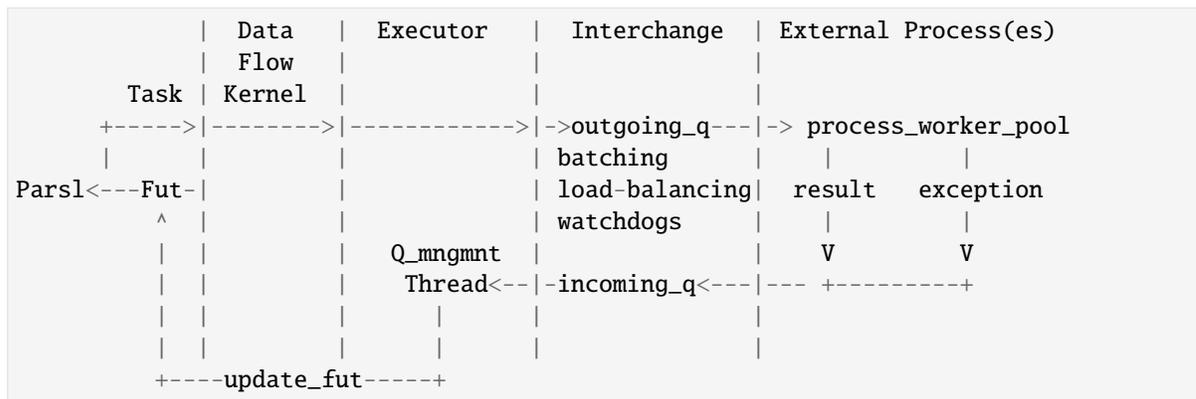
ExecutionProvider =
LocalProvider(channel=LocalChannel(envs={},
script_dir=None, user-
home='/home/docs/checkouts/readthedocs.org/user_builds/parsl/checkouts/l
cmd_timeout=30, init_blocks=1,
launcher=SingleNodeLauncher(debug=True,
fail_on_any=False), max_blocks=1, min_blocks=0,
move_files=None, nodes_per_block=1, parallelism=1,
worker_init="), launch_cmd: str | None = None, address:
str | None = None, worker_ports: Tuple[int, int] | None =
None, worker_port_range: Tuple[int, int] | None =
(54000, 55000), interchange_port_range: Tuple[int, int] |
None = (55000, 56000), storage_access: List[Staging] |
None = None, working_dir: str | None = None,
worker_debug: bool = False, cores_per_worker: float =
1.0, mem_per_worker: float | None = None, max_workers:
int | float | None = None, max_workers_per_node: int |
float | None = None, cpu_affinity: str = 'none',
available_accelerators: int | Sequence[str] = (),
prefetch_capacity: int = 0, heartbeat_threshold: int =
120, heartbeat_period: int = 30, drain_period: int | None
= None, poll_period: int = 10, address_probe_timeout:
int | None = None, worker_logdir_root: str | None =
None, enable_mpi_mode: bool = False, mpi_launcher: str
= 'mpiexec', block_error_handler: bool |
Callable[[BlockProviderExecutor, Dict[str, JobStatus]],
None] = True, encrypted: bool = False)
    
```

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



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 field expects an IPv4 address (`xxx.xxx.xxx.xxx`). Most login nodes on clusters have several network interfaces available, only some of which can be reached from the compute nodes. This field can be used to limit the executor to listen only on a specific interface, and limiting connections to the internal network. 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.
- **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*) – Deprecated. Please use `max_workers_per_node` instead.
- **max_workers_per_node** (*int*) – Caps the number of workers launched per node. `Default: None`
- **cpu_affinity** (*string*) – Whether or how each worker process sets thread affinity. Options include “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). The “block-reverse” option assigns adjacent cores to workers, but assigns the CPUs with large indices to low index workers (ex: assign 2-3 to worker 1, 0,1 to worker 2)

- **available_accelerators** (*int* / *list*) – Accelerators available for workers to use. Each worker will be pinned to exactly one of the provided accelerators, and no more workers will be launched than the number of accelerators.

Either provide the list of accelerator names or the number available. If a number is provided, Parsl will create names as integers starting with 0.

default: empty list

- **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
- **drain_period** (*int*) – The number of seconds after start when workers will begin to drain and then exit. Set this to a time that is slightly less than the maximum walltime of batch jobs to avoid killing tasks while they execute. For example, you could set this to the walltime minus a grace period for the batch job to start the workers, minus the expected maximum length of an individual task.
- **worker_logdir_root** (*string*) – In case of a remote file system, specify the path to where logs will be kept.
- **enable_mpi_mode** (*bool*) – If enabled, MPI launch prefixes will be composed for the batch scheduler based on the nodes available in each batch job and the resource_specification dict passed from the app. This is an experimental feature, please refer to the following doc section before use: https://parsl.readthedocs.io/en/stable/userguide/mpi_apps.html
- **mpi_launcher** (*str*) – This field is only used if enable_mpi_mode is set. Select one from the list of supported MPI launchers = (“srun”, “aprun”, “mpiexec”). default: “mpiexec”
- **encrypted** (*bool*) – Flag to enable/disable encryption (CurveZMQ). Default is False.

```

__init__(label: str = 'HighThroughputExecutor', provider: ExecutionProvider =
    LocalProvider(channel=LocalChannel(envs={}, script_dir=None,
    userhome='/home/docs/checkouts/readthedocs.org/user_builds/parsl/checkouts/latest/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, worker_init=""), launch_cmd: str | None = None, address: str | None = None,
    worker_ports: Tuple[int, int] | None = None, worker_port_range: Tuple[int, int] | None = (54000,
    55000), interchange_port_range: Tuple[int, int] | None = (55000, 56000), storage_access:
    List[Staging] | None = None, working_dir: str | None = None, worker_debug: bool = False,
    cores_per_worker: float = 1.0, mem_per_worker: float | None = None, max_workers: int | float |
    None = None, max_workers_per_node: int | float | None = None, cpu_affinity: str = 'none',
    available_accelerators: int | Sequence[str] = (), prefetch_capacity: int = 0, heartbeat_threshold:
    int = 120, heartbeat_period: int = 30, drain_period: int | None = None, poll_period: int = 10,
    address_probe_timeout: int | None = None, worker_logdir_root: str | None = None,
    enable_mpi_mode: bool = False, mpi_launcher: str = 'mpiexec', block_error_handler: bool |
    Callable[[BlockProviderExecutor, Dict[str, JobStatus]], None] = True, encrypted: bool = False)

```

Methods

<code>__init__([label, provider, launch_cmd, ...])</code>	
<code>connected_blocks()</code>	List of connected block ids
<code>connected_managers()</code>	Returns a list of dicts one for each connected managers.
<code>create_monitoring_info(status)</code>	Create a monitoring message for each block based on the poll status.
<code>get_usage_information()</code>	
<code>handle_errors(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 scale out the initial blocks.
<code>monitor_resources()</code>	Should resource monitoring happen for tasks on running on this executor?
<code>poll_facade()</code>	
<code>scale_in(blocks[, max_idletime])</code>	Scale in the number of active blocks by specified amount.
<code>scale_in_facade(n[, max_idletime])</code>	
<code>scale_out([blocks])</code>	Scales out the number of blocks by "blocks"
<code>scale_out_facade(n)</code>	
<code>send_monitoring_info(status)</code>	
<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([timeout])</code>	Shutdown the executor, including the interchange.
<code>start()</code>	Create the Interchange process and connect to it.
<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 outgoing_q.

Attributes

<code>bad_state_is_set</code>	Returns true if this executor is in an irrecoverable error state.
<code>connected_workers</code>	Returns the count of workers across all connected managers
<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>label</code>	
<code>logdir</code>	
<code>max_workers</code>	
<code>monitoring_radio</code>	Local radio for sending monitoring messages
<code>outstanding</code>	Returns the count of tasks outstanding across the interchange and managers
<code>provider</code>	
<code>radio_mode</code>	
<code>run_dir</code>	Path to the run directory.
<code>run_id</code>	UUID for the enclosing DFK.
<code>status_facade</code>	Return the status of all jobs/blocks of the executor of this poller.
<code>status_polling_interval</code>	Returns the interval, in seconds, at which the status method should be called.
<code>tasks</code>	
<code>worker_logdir</code>	
<code>workers_per_node</code>	

connected_blocks() → List[str]

List of connected block ids

connected_managers() → List[Dict[str, Any]]

Returns a list of dicts one for each connected managers. The dict contains info on manager(str:manager_id), block_id, worker_count, tasks(int), idle_durations(float), active(bool)

property connected_workers: int

Returns the count of workers across all connected managers

get_usage_information()

hold_worker(worker_id: str) → None

Puts a worker on hold, preventing scheduling of additional tasks to it.

This is called “hold” mostly because this only stops scheduling of tasks, and does not actually kill the worker.

Parameters

worker_id (*str*) – Worker id to be put on hold

initialize_scaling()

Compose the launch command and scale out the initial blocks.

property logdir

property max_workers

property outstanding: int

Returns the count of tasks outstanding across the interchange and managers

radio_mode: str = 'htex'

scale_in(blocks: int, max_idletime: float | None = None) → List[str]

Scale in the number of active blocks by specified amount.

The scale in method here is very rude. It doesn't give the workers the opportunity to finish current tasks or cleanup. This is tracked in issue #530

Parameters

- **blocks** (*int*) – Number of blocks to terminate and scale_in by
- **max_idletime** (*float*) – A time to indicate how long a block should be idle to be a candidate for scaling in.

If None then blocks will be force scaled in even if they are busy.

If a float, then only idle blocks will be terminated, which may be less than the requested number.

Return type

List of block IDs scaled in

shutdown(timeout: float = 10.0)

Shutdown the executor, including the interchange. This does not shut down any workers directly - workers should be terminated by the scaling mechanism or by heartbeat timeout.

Parameters

timeout (*float*) – Amount of time to wait for the Interchange process to terminate before we forcefully kill it.

start()

Create the Interchange process and connect to it.

status() → Dict[str, JobStatus]

Return the status of all jobs/blocks currently known to this executor.

Returns

a dictionary mapping block ids (in string) to job status

submit(func, resource_specification, *args, **kwargs)

Submits work to the outgoing_q.

The outgoing_q is an external process listens on this queue for new work. This method behaves like a submit call as described here [Python docs](#):

Parameters

- **func** (-) – Callable function

- **resource_specification** (-) – Dictionary containing relevant info about task that is needed by underlying executors.
- **args** (-) – List of arbitrary positional arguments.

Kwargs:

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

Returns

Future

property `worker_logdir`**property** `workers_per_node`: `int` | `float`**parsl.executors.WorkQueueExecutor**

```
class parsl.executors.WorkQueueExecutor(label: str = 'WorkQueueExecutor', provider: ExecutionProvider
    = LocalProvider(channel=LocalChannel(envs={}),
    script_dir=None, user_home='/home/docs/checkouts/readthedocs.org/user_builds/parsl/checkouts/latest/d
    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,
    worker_init=""), working_dir: str = '.', project_name: str | None
    = None, project_password_file: str | None = None, address: str |
    None = None, port: int = 0, env: Dict | None = None, shared_fs:
    bool = False, storage_access: List[Staging] | None = None,
    use_cache: bool = False, source: bool = False, pack: bool =
    False, extra_pkgs: List[str] | None = None, autolabel: bool =
    False, autolabel_window: int = 1, autocategory: bool = True,
    max_retries: int = 1, init_command: str = "", worker_options:
    str = "", full_debug: bool = True, worker_executable: str =
    'work_queue_worker', function_dir: str | None = None,
    coprocess: bool = False)
```

Executor to use Work Queue batch system

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

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.
- **project_name** (*str*) – If a project_name is given, then Work Queue will periodically report its status and performance back to the global WQ catalog, which can be viewed here: <http://ccl.cse.nd.edu/software/workqueue/status> 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 connect to Parsl using this port.

If 0, Work Queue will allocate a port number automatically. In this case, environment variables can be used to influence the choice of port, documented here: https://ccl.cse.nd.edu/software/manuals/api/html/work__queue_8h.html#a21714a10bcdcf5c3bd44a96f5dcbda6
 Default: `WORK_QUEUE_DEFAULT_PORT`.

- **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.
- **max_retries** (*int*) – Set the number of retries that Work Queue will make when a task fails. This is distinct from Parsl level retries configured in `parsl.config.Config`. Set to None to allow Work Queue to retry tasks forever. By default, this is set to 1, so that all retries will be managed by Parsl.
- **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 `''`.
- **worker_executable** (*str*) – The command used to invoke `work_queue_worker`. This can be used when the worker needs to be wrapped inside some other command (for example, to run the worker inside a container). Default is `'work_queue_worker'`.
- **function_dir** (*str*) – The directory where serialized function invocations are placed to be sent to workers. If undefined, this defaults to a directory under `runinfo/`. If `shared_filesystem=True`, then this directory must be visible from both the submitting side and workers.
- **coprocess** (*bool*) – Use Work Queue’s coprocess facility to avoid launching a new Python process for each task. Experimental. This requires a version of Work Queue / cctools after commit `874df524516441da531b694afc9d591e8b134b73` (release 7.5.0 is too early). Default is `False`.

```

__init__(label: str = 'WorkQueueExecutor', provider: ExecutionProvider =
LocalProvider(channel=LocalChannel(envs={}, script_dir=None,
userhome='/home/docs/checkouts/readthedocs.org/user_builds/parsl/checkouts/latest/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, worker_init=""), working_dir: str = '.', project_name: str | None = None,
project_password_file: str | None = None, address: str | None = None, port: int = 0, env: Dict |
None = None, shared_fs: bool = False, storage_access: List[Staging] | None = None, use_cache:
bool = False, source: bool = False, pack: bool = False, extra_pkgs: List[str] | None = None,
autolabel: bool = False, autolabel_window: int = 1, autocategory: bool = True, max_retries: int =
1, init_command: str = "", worker_options: str = "", full_debug: bool = True, worker_executable:
str = 'work_queue_worker', function_dir: str | None = None, coprocess: bool = False)

```

Methods

<code>__init__([label, provider, working_dir, ...])</code>	
<code>atexit_cleanup()</code>	
<code>create_monitoring_info(status)</code>	Create a monitoring message for each block based on the poll status.
<code>handle_errors(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>poll_facade()</code>	
<code>scale_in(count)</code>	Scale in method.
<code>scale_in_facade(n[, max_idletime])</code>	
<code>scale_out([blocks])</code>	Scales out the number of blocks by "blocks"
<code>scale_out_facade(n)</code>	
<code>send_monitoring_info(status)</code>	
<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>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>label</code>	
<code>monitoring_radio</code>	Local radio for sending monitoring messages
<code>outstanding</code>	Count the number of outstanding tasks.
<code>provider</code>	
<code>radio_mode</code>	
<code>run_dir</code>	Path to the run directory.
<code>run_id</code>	UUID for the enclosing DFK.
<code>status_facade</code>	Return the status of all jobs/blocks of the executor of this poller.
<code>status_polling_interval</code>	Returns the interval, in seconds, at which the status method should be called.
<code>tasks</code>	
<code>workers_per_node</code>	

`atexit_cleanup()`

`initialize_scaling()`

Compose the launch command and call scale out

Scales the workers to the appropriate nodes with provider

property outstanding: int

Count the number of outstanding tasks. This is inefficiently implemented and probably could be replaced with a counter.

radio_mode: str = 'filesystem'

`scale_in(count: int) → List[str]`

Scale in method.

`shutdown(*args, **kwargs)`

Shutdown the executor. Sets flag to cancel the submit process and collector thread, which shuts down the Work Queue system submission.

`start()`

Create submit process and collector thread to create, send, and retrieve Parsl tasks within the Work Queue system.

`submit(func, resource_specification, *args, **kwargs)`

Processes the Parsl app by its arguments and submits the function information to the task queue, to be executed using the Work Queue system. The args and kwargs are processed for input and output files to the Parsl app, so that the files are appropriately specified for the Work Queue task.

Parameters

- **func** (*function*) – Parsl app to be submitted to the Work Queue system
- **args** (*list*) – Arguments to the Parsl app
- **kwargs** (*dict*) – Keyword arguments to the Parsl app

property **workers_per_node**: **int** | **float**

parsl.executors.taskvine.TaskVineExecutor

```
class parsl.executors.taskvine.TaskVineExecutor(label: str = 'TaskVineExecutor',
                                               worker_launch_method: Literal['provider'] |
                                               Literal['factory'] | Literal['manual'] = 'factory',
                                               function_exec_mode: Literal['regular'] |
                                               Literal['serverless'] = 'regular', manager_config:
                                               TaskVineManagerConfig =
                                               TaskVineManagerConfig(port=0, address=None,
                                               project_name=None, project_password_file=None,
                                               env_vars=None, init_command="", env_pack=None,
                                               app_pack=False, extra_pkgs=None, max_retries=1,
                                               library_config=None, shared_fs=False,
                                               autolabel=False, autolabel_algorithm='max-xput',
                                               autolabel_window=None, autcategory=True,
                                               enable_peer_transfers=True,
                                               wait_for_workers=None, vine_log_dir=None),
                                               factory_config: TaskVineFactoryConfig =
                                               TaskVineFactoryConfig(factory_timeout=300,
                                               scratch_dir=None, min_workers=1, max_workers=1,
                                               workers_per_cycle=1, worker_options=None,
                                               worker_executable='vine_worker',
                                               worker_timeout=300, cores=None, gpus=None,
                                               memory=None, disk=None, python_env=None,
                                               batch_type='local', condor_requirements=None,
                                               batch_options=None, _project_port=0,
                                               _project_address=None, _project_name=None,
                                               _project_password_file=None), provider:
                                               ExecutionProvider | None =
                                               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, worker_init=""), storage_access:
                                               List[Staging] | None = None)
```

Executor to use TaskVine dynamic workflow system

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

This Executor sets up configurations for the TaskVine manager, TaskVine factory, and run both in separate processes. Sending tasks and receiving results are done through multiprocessing module native to Python.

Parameters

- **label** (*str*) – A human readable label for the executor, unique with respect to other executors. Default is “TaskVineExecutor”.
- **worker_launch_method** (*Union[Literal['provider'], Literal['factory'], Literal['manual']]*) – Choose to use Parsl provider, TaskVine factory, or manual user-provided workers to scale workers. Options are among {'provider', 'factory', 'manual'}. Default is 'factory'.
- **function_exec_mode** (*Union[Literal['regular'], Literal['serverless']]*) – Choose to execute functions with a regular fresh python process or a pre-warmed forked python process. Default is 'regular'.
- **manager_config** (*TaskVineManagerConfig*) – Configuration for the TaskVine manager. Default
- **factory_config** (*TaskVineFactoryConfig*) – Configuration for the TaskVine factory. Use of factory is disabled by default.
- **provider** (*ExecutionProvider*) – The Parsl provider that will spawn worker processes. Default to spawning one local vine worker process.
- **storage_access** (*List[Staging]*) – Define Parsl file staging providers for this executor. Default is None.

```
__init__(label: str = 'TaskVineExecutor', worker_launch_method: Literal['provider'] | Literal['factory'] |
        Literal['manual'] = 'factory', function_exec_mode: Literal['regular'] | Literal['serverless'] =
        'regular', manager_config: TaskVineManagerConfig = TaskVineManagerConfig(port=0,
        address=None, project_name=None, project_password_file=None, env_vars=None,
        init_command="", env_pack=None, app_pack=False, extra_pkgs=None, max_retries=1,
        library_config=None, shared_fs=False, autolabel=False, autolabel_algorithm='max-xput',
        autolabel_window=None, autocategory=True, enable_peer_transfers=True,
        wait_for_workers=None, vine_log_dir=None), factory_config: TaskVineFactoryConfig =
        TaskVineFactoryConfig(factory_timeout=300, scratch_dir=None, min_workers=1,
        max_workers=1, workers_per_cycle=1, worker_options=None, worker_executable='vine_worker',
        worker_timeout=300, cores=None, gpus=None, memory=None, disk=None, python_env=None,
        batch_type='local', condor_requirements=None, batch_options=None, _project_port=0,
        _project_address=None, _project_name=None, _project_password_file=None), provider:
        ExecutionProvider | None = LocalProvider(channel=LocalChannel(envs={}, script_dir=None,
        userhome='/home/docs/checkouts/readthedocs.org/user_builds/parsl/checkouts/latest/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, worker_init=""), storage_access: List[Staging] | None = None)
```

Methods

<code>__init__([label, worker_launch_method, ...])</code>	
<code>atexit_cleanup()</code>	
<code>create_monitoring_info(status)</code>	Create a monitoring message for each block based on the poll status.
<code>handle_errors(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>poll_facade()</code>	
<code>scale_in(count)</code>	Scale in method.
<code>scale_in_facade(n[, max_idletime])</code>	
<code>scale_out([blocks])</code>	Scales out the number of blocks by "blocks"
<code>scale_out_facade(n)</code>	
<code>send_monitoring_info(status)</code>	
<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 TaskVine 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 TaskVine system.

Attributes

<code>bad_state_is_set</code>	Returns true if this executor is in an irrecoverable error state.
<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>label</code>	
<code>monitoring_radio</code>	Local radio for sending monitoring messages
<code>outstanding</code>	Count the number of outstanding tasks.
<code>provider</code>	
<code>radio_mode</code>	
<code>run_dir</code>	Path to the run directory.
<code>run_id</code>	UUID for the enclosing DFK.
<code>status_facade</code>	Return the status of all jobs/blocks of the executor of this poller.
<code>status_polling_interval</code>	Returns the interval, in seconds, at which the status method should be called.
<code>tasks</code>	
<code>workers_per_node</code>	

atexit_cleanup()

initialize_scaling()

Compose the launch command and call scale out

Scales the workers to the appropriate nodes with provider

property outstanding: int

Count the number of outstanding tasks.

radio_mode: str = 'filesystem'

scale_in(count: int) → List[str]

Scale in method. Cancel a given number of blocks

shutdown(*args, **kwargs)

Shutdown the executor. Sets flag to cancel the submit process and collector thread, which shuts down the TaskVine system submission.

start()

Create submit process and collector thread to create, send, and retrieve Parsl tasks within the TaskVine system.

submit(func, resource_specification, *args, **kwargs)

Processes the Parsl app by its arguments and submits the function information to the task queue, to be executed using the TaskVine system. The args and kwargs are processed for input and output files to the Parsl app, so that the files are appropriately specified for the TaskVine task.

Parameters

- **func** (*function*) – Parsl app to be submitted to the TaskVine system
- **resource_specification** (*dict*) – Dictionary containing relevant info about task. Include information about resources of task, execution mode of task (out of {regular, serverless}).
- **args** (*list*) – Arguments to the Parsl app
- **kwargs** (*dict*) – Keyword arguments to the Parsl app

property workers_per_node: `int` | `float`

parsl.executors.FluxExecutor

```
class parsl.executors.FluxExecutor(provider: ExecutionProvider | None = None, working_dir: str | None = None, label: str = 'FluxExecutor', flux_executor_kwargs: Mapping = {}, flux_path: str | None = None, launch_cmd: str | None = None)
```

Executor that uses Flux to schedule and run jobs.

Every callable submitted to the executor is wrapped into a Flux job.

This executor requires that there be a Flux installation available locally, and that it can be located either in PATH or through the `flux_path` argument.

Flux jobs are fairly heavyweight. As of Flux v0.25, a single Flux instance is (on many systems) capped at 50 jobs per second. As such, this executor is not a good fit for use-cases consisting of large numbers of small, fast jobs.

However, Flux is great at handling jobs with large resource requirements, and collections of jobs with varying resource requirements.

Note that due to vendor-specific extensions, on certain Cray machines like ALCF's Theta or LANL's Trinity/Trinity, Flux cannot run applications that use the default MPI library. Generally the only workaround is to recompile with another MPI library like OpenMPI.

This executor acts as a sort of wrapper around a `flux.job.FluxExecutor`, which can be confusing since both wrapped and wrapper classes share the same name. Whenever possible, the underlying executor is referred by its fully qualified name, `flux.job.FluxExecutor`.

Parameters

- **working_dir** (*str*) – Directory in which the executor should place its files, possibly overwriting existing files. If `None`, generate a unique directory.
- **label** (*str*) – Label for this executor instance.
- **flux_handle_args** (*collections.abc.Sequence*) – Positional arguments to `flux.Flux()` instance, if any. The first positional argument, `url`, is provided by this executor.
- **flux_executor_kwargs** (*collections.abc.Mapping*) – Keyword arguments to pass to the underlying `flux.job.FluxExecutor()` instance, if any. Note that the `handle_args` keyword argument is provided by this executor, in order to supply the URL of a remote Flux instance.
- **flux_path** (*str*) – Path to flux installation to use, or `None` to search PATH for flux.
- **launch_cmd** (*str*) – The command to use when launching the executor's backend. The default command is available as the `DEFAULT_LAUNCH_COMMAND` attribute. The default command starts a new Flux instance, which may not be desirable if a Flux instance will already be provisioned (this is not likely).

```
__init__(provider: ExecutionProvider | None = None, working_dir: str | None = None, label: str =
'FluxExecutor', flux_executor_kwargs: Mapping = {}, flux_path: str | None = None, launch_cmd:
str | None = None)
```

Methods

<code>__init__([provider, working_dir, label, ...])</code>	
<code>monitor_resources()</code>	Should resource monitoring happen for tasks on running on this executor?
<code>shutdown([wait])</code>	Shut down the executor, causing further calls to <code>submit</code> to fail.
<code>start()</code>	Called when DFK starts the executor when the config is loaded.
<code>submit(func, resource_specification, *args, ...)</code>	Wrap a callable in a Flux job and submit it to Flux.

Attributes

<code>DEFAULT_LAUNCH_CMD</code>	
<code>hub_address</code>	Address to the Hub for monitoring.
<code>hub_port</code>	Port to the Hub for monitoring.
<code>label</code>	
<code>monitoring_radio</code>	Local radio for sending monitoring messages
<code>radio_mode</code>	
<code>run_dir</code>	Path to the run directory.
<code>run_id</code>	UUID for the enclosing DFK.

```
DEFAULT_LAUNCH_CMD = '{flux} start {python} {manager} {protocol} {hostname} {port}'
```

`shutdown(wait=True)`

Shut down the executor, causing further calls to `submit` to fail.

Parameters

wait – If True, do not return until all submitted Futures are done.

`start()`

Called when DFK starts the executor when the config is loaded.

`submit(func: Callable, resource_specification: Dict[str, Any], *args: Any, **kwargs: Any)`

Wrap a callable in a Flux job and submit it to Flux.

Parameters

- **func** – The callable to submit as a job to Flux
- **resource_specification** – A mapping defining the resources to allocate to the Flux job.

Only the following keys are checked for:

- num_tasks: the number of tasks to launch (MPI ranks for an MPI job), default 1
- cores_per_task: cores per task, default 1
- gpus_per_task: gpus per task, default 1
- num_nodes: if > 0, evenly distribute the allocated cores/gpus across the given number of nodes. Does *not* give the job exclusive access to those nodes; this option only affects distribution.
- **args** – positional arguments for the callable
- **kwargs** – keyword arguments for the callable

parsl.executors.radical.RadicalPilotExecutor

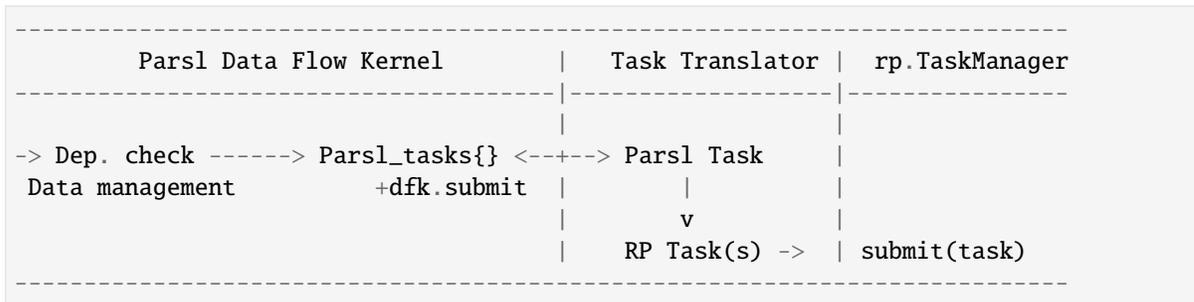
```
class parsl.executors.radical.RadicalPilotExecutor(resource: str, label: str = 'RPEX', bulk_mode:
    bool = False, working_dir: str | None = None,
    rpex_cfg: ResourceConfig | None = None,
    **rpex_pilot_kwargs)
```

Executor is designed for executing heterogeneous tasks
in terms of type/resource.

The RadicalPilotExecutor system has the following main components:

1. “start”: Create and start the RADICAL-Pilot runtime components `rp.Session`, `rp.PilotManager` and `rp.TaskManager`.
2. “translate”: Unwrap, identify, and parse Parsl apps into `rp.TaskDescription`.
3. “submit”: Submit Parsl apps to `rp.TaskManager`.
4. “shut_down”: Shut down the RADICAL-Pilot runtime and all associated components.

Here is a diagram



The RadicalPilotExecutor creates a `rp.Session`, `rp.TaskManager`, and `rp.PilotManager`. The executor receives the Parsl apps from the DFK and translates these apps (in-memory) into `rp.TaskDescription` object to be passed to the `rp.TaskManager`. This executor has two submission mechanisms:

1. `Default_mode`: where the executor submits the tasks directly to RADICAL-Pilot.
2. `Bulk_mode`: where the executor accumulates N tasks (functions and executables) and submit them.

Parameters

- **rpex_cfg** – a dataclass specifying resource configuration. Default is `ResourceConfig` instance.
- **label** (*str*) – Label for this executor instance. Default is “RPEX”.

- **bulk_mode** (*bool*) – Enable bulk mode submission and execution. Default is False (stream).
- **resource** (*Optional[str]*) – The resource name of the targeted HPC machine or cluster. Default is local.localhost (user local machine).
- **runtime** (*int*) – The maximum runtime for the entire job in minutes. Default is 30.
- **working_dir** (*str*) – The working dir to be used by the executor.
- **rpex_pilot_kwargs** (*Dict of kwargs that are passed directly to the rp.PilotDescription object.*) –
- **information** (*For more*) –

`__init__(resource: str, label: str = 'RPEX', bulk_mode: bool = False, working_dir: str | None = None, rpex_cfg: ResourceConfig | None = None, **rpex_pilot_kwargs)`

Methods

<code>__init__(resource[, label, bulk_mode, ...])</code>	
<code>monitor_resources()</code>	Should resource monitoring happen for tasks on running on this executor?
<code>shutdown([hub, targets, block])</code>	Shutdown the executor, including all RADICAL-Pilot components.
<code>start()</code>	Create the Pilot component and pass it.
<code>submit(func, resource_specification, *args, ...)</code>	Submits tasks in stream mode or bulks (bulk mode) to RADICAL-Pilot rp.TaskManager.
<code>task_state_cb(task, state)</code>	Update the state of Parsl Future apps Based on RP task state callbacks.
<code>task_translate(tid, func, ...)</code>	Convert Parsl function to RADICAL-Pilot rp.TaskDescription
<code>unwrap(func, args)</code>	Unwrap a Parsl app and its args for further processing.

Attributes

<code>hub_address</code>	Address to the Hub for monitoring.
<code>hub_port</code>	Port to the Hub for monitoring.
<code>label</code>	
<code>monitoring_radio</code>	Local radio for sending monitoring messages
<code>radio_mode</code>	
<code>run_dir</code>	Path to the run directory.
<code>run_id</code>	UUID for the enclosing DFK.

shutdown(*hub=True, targets='all', block=False*)

Shutdown the executor, including all RADICAL-Pilot components.

start()

Create the Pilot component and pass it.

submit(*func, resource_specification, *args, **kwargs*)

Submits tasks in stream mode or bulks (bulk mode) to RADICAL-Pilot rp.TaskManager.

task_state_cb(*task, state*)

Update the state of Parsl Future apps Based on RP task state callbacks.

task_translate(*tid, func, parsl_resource_specification, args, kwargs*)

Convert Parsl function to RADICAL-Pilot rp.TaskDescription

unwrap(*func, args*)

Unwrap a Parsl app and its args for further processing.

Parameters

- **func** (*callable*) – The function to be unwrapped.
- **args** (*tuple*) – The arguments associated with the function.

Returns

A tuple containing the unwrapped function, adjusted arguments, and task type information.

Return type

tuple

Launchers

<i>parsl.launchers.base.Launcher</i>	Launchers are basically wrappers for user submitted scripts as they are submitted to a specific execution resource.
<i>parsl.launchers.SimpleLauncher</i>	Does no wrapping.
<i>parsl.launchers.SingleNodeLauncher</i>	Worker launcher that wraps the user's command with the framework to launch multiple command invocations in parallel.
<i>parsl.launchers.SrunLauncher</i>	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.
<i>parsl.launchers.AprunLauncher</i>	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
<i>parsl.launchers.SrunMPILauncher</i>	Launches as many workers as MPI tasks to be executed concurrently within a block.
<i>parsl.launchers.GnuParallelLauncher</i>	Worker launcher that wraps the user's command with the framework to launch multiple command invocations via GNU parallel sshlogin.
<i>parsl.launchers.MpiExecLauncher</i>	Worker launcher that wraps the user's command with the framework to launch multiple command invocations via mpiexec.
<i>parsl.launchers.JsrunLauncher</i>	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
<i>parsl.launchers.WrappedLauncher</i>	Wraps the command by prepending commands before a user's command

parsl.launchers.base.Launcher

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

Launchers are basically wrappers for user submitted scripts as they are submitted to a specific execution resource.

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

Methods

```
__init__([debug])
```

parsl.launchers.SimpleLauncher

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

Does no wrapping. Just returns the command as-is

`__init__`(*debug: bool = True*) → None

Methods

```
__init__([debug])
```

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*)

Methods

```
__init__([debug, fail_on_any])
```

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

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

param overrides

This string will be passed to the srun launcher. Default: ""

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

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

param overrides

This string will be passed to the aprun launcher. Default: ""

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: ""

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)
```

Methods

```
__init__([debug])
```

parsl.launchers.MpiExecLauncher

```
class parsl.launchers.MpiExecLauncher(debug: bool = True, bind_cmd: str = '--bind-to', overrides: str = "")
```

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, bind_cmd: str = '--bind-to', overrides: str = "")
```

Parameters

- **bind_cmd** (*str*) – Name of the argument for binding ranks to CPUs
- **overrides** (*str*) – Additional arguments to add to the invocation

Methods

<code>__init__</code> ([debug, bind_cmd, overrides])	<p>param bind_cmd Name of the argument for binding ranks to CPUs</p>
--	---

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

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., <code>time</code>)
---	---

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.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.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.

`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)`

Methods

<code>__init__</code> (channels, worker_init, ...)	
<code>cancel</code> (job_ids)	Cancel a list of jobs with job_ids
<code>status</code> (job_ids)	Get status of the list of jobs with job_ids
<code>submit</code> (command, tasks_per_node[, job_name])	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>status_polling_interval</code>	Returns the interval, in seconds, at which the status method should be called.

`cancel`(job_ids)

Cancel a list of jobs with job_ids

Parameters

job_ids (*list of strings*) – List of job id strings

Returns

list of confirmation bools

Return type

[True, False...]

property `label`

Provides the label for this provider

`status`(job_ids)

Get status of the list of jobs with job_ids

Parameters

job_ids (*list of strings*) – List of job id strings

Return type

list of JobStatus objects

property `status_polling_interval`

Returns the interval, in seconds, at which the status method should be called.

Returns

the number of seconds to wait between calls to status()

submit(*command*, *tasks_per_node*, *job_name*='parsl.adhoc')

Submits the command onto a channel from the list of channels

Submit returns an ID that corresponds to the task that was just submitted.

Parameters

- **command** (*(String)*) – Commandline invocation to be made on the remote side.
- **tasks_per_node** (*(int)*) – command invocations to be launched per node
- **job_name** (*(String)*) – Name of the job. Default : parsl.adhoc

Returns

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

parsl.providers.AWSPProvider

```
class parsl.providers.AWSPProvider(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))
```

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. With AWS, usually the default *SingleNodeLauncher* will be appropriate.
- **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))
```

Methods

<code>__init__(image_id, key_name[, init_blocks, ...])</code>	
<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.
<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.

`cancel(job_ids)`

Cancel the jobs specified by a list of job ids.

Parameters

`job_ids` (*list of str*) – List of job identifiers

Returns

Each entry in the list will contain False if the operation fails. Otherwise, the entry will be True.

Return type

list of bool

`config_route_table(vpc, internet_gateway)`

Configure route table for Virtual Private Cloud (VPC).

Parameters

- `vpc` (*dict*) – Representation of the VPC (created by `create_vpc()`).
- `internet_gateway` (*dict*) – Representation of the internet gateway (created by `create_vpc()`).

`create_name_tag_spec(resource_type, name)`

Create a new tag specification for a resource name.

Parameters

- `resource_type` (*str*) – The AWS resource type
- `name` (*str*) – The name to assign to the resource

Returns

A TagSpecifications record to be passed into the creation of a new AWS resource.

Return type

record

`create_session()`

Create a session.

First we look in `self.key_file` for a path to a json file with the credentials. The key file should have 'AWSAccessKeyId' and 'AWSSecretKey'.

Next we look at `self.profile` for a profile name and try to use the Session call to automatically pick up the keys for the profile from the user default keys file `~/.aws/config`.

Finally, boto3 will look for the keys in environment variables: `AWS_ACCESS_KEY_ID`: The access key for your AWS account. `AWS_SECRET_ACCESS_KEY`: The secret key for your AWS account. `AWS_SESSION_TOKEN`: The session key for your AWS account. This is only needed when you are using temporary credentials. The `AWS_SECURITY_TOKEN` environment variable can also be used, but is only

supported for backwards compatibility purposes. `AWS_SESSION_TOKEN` is supported by multiple AWS SDKs besides python.

create_vpc()

Create and configure VPC

We create a VPC with CIDR 10.0.0.0/16, which provides up to 64,000 instances.

We attach a subnet for each availability zone within the region specified in the config. We give each subnet an ip range like 10.0.X.0/20, which is large enough for approx. 4000 instances.

Security groups are configured in function `security_group`.

generate_aws_id()

Generate a new ID for AWS resources.

Returns

An ID of the form 'parsl.aws.123456.789' for giving resources unique identifiers.

Return type

str

get_instance_state(*instances=None*)

Get states of all instances on EC2 which were started by this file.

goodbye()

initialize_boto_client()

Initialize the boto client.

property label

Provides the label for this provider

read_state_file(*state_file*)

Read the state file, if it exists.

If this script has been run previously, resource IDs will have been written to a state file. On starting a run, a state file will be looked for before creating new infrastructure. Information on VPCs, security groups, and subnets are saved, as well as running instances and their states.

AWS has a maximum number of VPCs per region per account, so we do not want to clutter users' AWS accounts with security groups and VPCs that will be used only once.

security_group(*vpc, name*)

Create and configure a new security group.

Allows all ICMP in, all TCP and UDP in within VPC.

This security group is very open. It allows all incoming ping requests on all ports. It also allows all outgoing traffic on all ports. This can be limited by changing the allowed port ranges.

Parameters

- **vpc** (*VPC instance*) – VPC in which to set up security group.
- **name** (*str*) – Name tag for the newly created security group.

show_summary()

Print human readable summary of current AWS state to log and to console.

shut_down_instance(*instances=None*)

Shut down a list of instances, if provided.

If no instance is provided, the last instance started up will be shut down.

spin_up_instance(*command, job_name*)

Start an instance in the VPC in the first available subnet.

N instances will be started if `nodes_per_block > 1`. Not supported. We only do 1 node per block.

Parameters

- **command** (*str*) – Command string to execute on the node.
- **job_name** (*str*) – Name associated with the instances.

status(*job_ids*)

Get the status of a list of jobs identified by their ids.

Parameters

job_ids (*list of str*) – Identifiers for the jobs.

Returns

The status codes of the requested jobs.

Return type

list of int

property status_polling_interval

Returns the interval, in seconds, at which the status method should be called.

Returns

the number of seconds to wait between calls to status()

submit(*command='sleep 1', tasks_per_node=1, job_name='parsl.aws'*)

Submit the command onto a freshly instantiated AWS EC2 instance.

Submit returns an ID that corresponds to the task that was just submitted.

Parameters

- **command** (*str*) – Command to be invoked on the remote side.
- **tasks_per_node** (*int (default=1)*) – Number of command invocations to be launched per node
- **job_name** (*str*) – Prefix for the job name.

Returns

If at capacity, None will be returned. Otherwise, the job identifier will be returned.

Return type

None or *str*

teardown()

Teardown the EC2 infrastructure.

Terminate all EC2 instances, delete all subnets, delete security group, delete VPC, and reset all instance variables.

write_state_file()

Save information that must persist to a file.

We do not want to create a new VPC and new identical security groups, so we save information about them in a file between runs.

`xstr(s)`

`parsl.providers.CobaltProvider`

```
class parsl.providers.CobaltProvider(channel=LocalChannel(envs={}, script_dir=None, user-  
home='/home/docs/checkouts/readthedocs.org/user_builds/parsl/checkouts/latest/docs',  
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

WARNING: CobaltProvider is deprecated and will be removed by 2024.04

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,  
userhome='/home/docs/checkouts/readthedocs.org/user_builds/parsl/checkouts/latest/docs'),  
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)
```

Methods

<code>__init__</code> ([channel, nodes_per_block, ...])	
<code>cancel</code> (job_ids)	Cancels the jobs specified by a list of job ids
<code>execute_wait</code> (cmd[, timeout])	
<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])	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.

`cancel`(job_ids)

Cancels the jobs specified by a list of job ids

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

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

property `status_polling_interval`

Returns the interval, in seconds, at which the status method should be called.

Returns

the number of seconds to wait between calls to status()

`submit`(command, tasks_per_node, job_name='parsl.cobalt')

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

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

If tasks_per_node == 1:

A single node is provisioned

If tasks_per_node > 1 :

tasks_per_node number of nodes are provisioned.

Parameters

- **command** (-) – (String) Commandline invocation to be made on the remote side.
- **tasks_per_node** (-) – command invocations to be launched per node

Kwargs:

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

Returns

At capacity, cannot provision more - job_id: (string) Identifier for the job

Return type

- None

parsl.providers.CondorProvider

```
class parsl.providers.CondorProvider(channel: Channel = LocalChannel(envs={}, script_dir=None, user_home='/home/docs/checkouts/readthedocs.org/user_builds/parsl/checkouts/latest/docs', nodes_per_block: int = 1, cores_per_slot: int | None = None, mem_per_slot: float | None = None, init_blocks: int = 1, min_blocks: int = 0, max_blocks: int = 1, parallelism: float = 1, environment: Dict[str, str] | None = None, project: str = "", scheduler_options: str = "", transfer_input_files: List[str] = [], walltime: str = '00:10:00', worker_init: str = "", launcher: Launcher = SingleNodeLauncher(debug=True, fail_on_any=False), requirements: str = "", cmd_timeout: int = 60, cmd_chunk_size: int = 100)
```

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.
- **cmd_chunk_size** (`int`) – Calls to the scheduler will be made for chunks of blocks with this size.

```
__init__(channel: Channel = LocalChannel(envs={}, script_dir=None,
userhome='/home/docs/checkouts/readthedocs.org/user_builds/parsl/checkouts/latest/docs'),
nodes_per_block: int = 1, cores_per_slot: int | None = None, mem_per_slot: float | None = None,
init_blocks: int = 1, min_blocks: int = 0, max_blocks: int = 1, parallelism: float = 1, environment:
Dict[str, str] | None = None, project: str = "", scheduler_options: str = "", transfer_input_files:
List[str] = [], walltime: str = '00:10:00', worker_init: str = "", launcher: Launcher =
SingleNodeLauncher(debug=True, fail_on_any=False), requirements: str = "", cmd_timeout: int =
60, cmd_chunk_size: int = 100) → None
```

Methods

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

`cancel`(job_ids)

Cancels the jobs specified by a list of job IDs.

Parameters

job_ids (*list of str*) – The job IDs to cancel.

Returns

Each entry in the list will be True if the job is cancelled successfully, otherwise False.

Return type

list of bool

`status`(job_ids)

Get the status of a list of jobs identified by their ids.

Parameters

job_ids (*list of int*) – Identifiers of jobs for which the status will be returned.

Returns

Status codes for the requested jobs.

Return type

List of `int`

property status_polling_interval

Returns the interval, in seconds, at which the status method should be called.

Returns

the number of seconds to wait between calls to `status()`

submit(*command, tasks_per_node, job_name='parsl.condor'*)

Submits the command onto an Local Resource Manager job.

example file with the complex case of multiple submits per job:

```
Universe =vanilla output = out.${Cluster}.${Process} error = err.${Cluster}.${Process} log =
log.${Cluster} leave_in_queue = true executable = test.sh queue 5 executable = foo queue 1
```

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

Parameters

- **command** (*str*) – Command to execute
- **job_name** (*str*) – Job name prefix.
- **tasks_per_node** (*int*) – command invocations to be launched per node

Returns

None if at capacity and cannot provision more; otherwise the identifier for the job.

Return type

None or `str`

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)
```

Methods

<code>__init__(project_id, key_file, region, ...)</code>	
<code>bye()</code>	
<code>cancel(job_ids)</code>	Cancels the resources identified by the job_ids 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>

bye()

cancel(*job_ids*)

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

Parameters

job_ids (-) – A list of job identifiers

Returns

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

Raises

- **ExecutionProviderException** or its subclasses -

`create_instance(command="")`

`delete_instance(name)`

`get_zone(region)`

`status(job_ids)`

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

Parameters

`job_ids` (-) – A list of job identifiers

Returns

- A list of JobStatus objects corresponding to each job_id in the job_ids list.

Raises

- **ExecutionProviderException** or its subclasses -

`property status_polling_interval`

`submit(command, tasks_per_node, job_name='parsl.gcs')`

The submit method takes the command string to be executed upon instantiation of a resource most often to start a pilot.

Args :

- `command` (str) : The bash command string to be executed.
- `tasks_per_node` (int) : command invocations to be launched per node

KWargs:

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

Returns

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

Raises

- **ExecutionProviderException** or its subclasses -

parsl.providers.GridEngineProvider

```
class parsl.providers.GridEngineProvider(channel=LocalChannel(envs={}, script_dir=None, user_home='/home/docs/checkouts/readthedocs.org/user_builds/parsl/checkouts/latest', 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,
userhome='/home/docs/checkouts/readthedocs.org/user_builds/parsl/checkouts/latest/docs'),
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)
```

Methods

<code>__init__</code> ([channel, nodes_per_block, ...])	
<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.

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.

`cancel(job_ids)`

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

Parameters

`job_ids` (-) – A list of job identifiers

Returns

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

Raises

– **ExecutionProviderException** or its subclasses –

`get_configs(command, tasks_per_node)`

Compose a dictionary with information for writing the submit script.

`property status_polling_interval`

Returns the interval, in seconds, at which the status method should be called.

Returns

the number of seconds to wait between calls to `status()`

`submit(command, tasks_per_node, job_name='parsl.sge')`

The submit method takes the command string to be executed upon instantiation of a resource most often to start a pilot.

Args :

- `command` (str) : The bash command string to be executed.
- `tasks_per_node` (int) : command invocations to be launched per node

KWargs:

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

Returns

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

Raises

– **ExecutionProviderException** or its subclasses –

parsl.providers.LocalProvider

```
class parsl.providers.LocalProvider(channel=LocalChannel(envs={}, script_dir=None, user-
    home='/home/docs/checkouts/readthedocs.org/user_builds/parsl/checkouts/latest/docs'),
    nodes_per_block=1, launcher=SingleNodeLauncher(debug=True,
    fail_on_any=False), init_blocks=1, min_blocks=0, max_blocks=1,
    worker_init="", cmd_timeout=30, parallelism=1, move_files=None)
```

Local Execution Provider

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

Parameters

- **min_blocks** (*int*) – Minimum number of blocks to maintain.
- **max_blocks** (*int*) – Maximum number of blocks to maintain.
- **parallelism** (*float*) – Ratio of provisioned task slots to active tasks. A parallelism value of 1 represents aggressive scaling where as many resources as possible are used; parallelism close to 0 represents the opposite situation in which as few resources as possible (i.e., min_blocks) are used.
- **move_files** (*Optional[Bool]*) – Should files be moved? By default, Parsl will try to figure this out itself (= None). If True, then will always move. If False, will never move.
- **worker_init** (*str*) – Command to be run before starting a worker, such as ‘module load Anaconda; source activate env’.

```
__init__(channel=LocalChannel(envs={}, script_dir=None,
    userhome='/home/docs/checkouts/readthedocs.org/user_builds/parsl/checkouts/latest/docs'),
    nodes_per_block=1, launcher=SingleNodeLauncher(debug=True, fail_on_any=False),
    init_blocks=1, min_blocks=0, max_blocks=1, worker_init="", cmd_timeout=30, parallelism=1,
    move_files=None)
```

Methods

<code>__init__([channel, nodes_per_block, ...])</code>	
<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.

cancel(*job_ids*)

Cancels the jobs specified by a list of job ids

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

Returns: [True] Always returns true for every job_id, regardless of whether an individual cancel failed (unless an exception is raised)

property label

Provides the label for this provider

status(*job_ids*)

Get the status of a list of jobs identified by their ids.

Parameters

job_ids (-) – List of identifiers for the jobs

Returns

- List of status codes.

property status_polling_interval

Returns the interval, in seconds, at which the status method should be called.

Returns

the number of seconds to wait between calls to status()

submit(*command, tasks_per_node, job_name='parsl.localprovider'*)

Submits the command onto an Local Resource Manager job. Submit returns an ID that corresponds to the task that was just submitted.

If tasks_per_node < 1:

1/tasks_per_node is provisioned

If tasks_per_node == 1:

A single node is provisioned

If tasks_per_node > 1 :

tasks_per_node nodes are provisioned.

Parameters

- **command** (-) – (String) Commandline invocation to be made on the remote side.
- **tasks_per_node** (-) – command invocations to be launched per node

Kwargs:

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

Returns

At capacity, cannot provision more - **job_id**: (string) Identifier for the job

Return type

- None

parsl.providers.LSFProvider

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

LSF Execution Provider

This provider uses bsub to submit, bjobs for status and bkill to cancel jobs. The bsub 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. When `request_by_nodes` is False, it is computed by `cores_per_block / cores_per_node`.
- **cores_per_block** (*int*) – Cores to provision per block. Enabled only when `request_by_nodes` is False.
- **cores_per_node** (*int*) – Cores to provision per node. Enabled only when `request_by_nodes` is False.
- **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
- **queue** (*str*) – Queue to which to submit the job request
- **scheduler_options** (*str*) – String to prepend to the #BSUB 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.) –
- **bsub_redirection** (*Bool*) – Should a redirection symbol “<” be included when submitting jobs, i.e., `Bsub < job_script`.

- **request_by_nodes** (*Bool*) – Request by nodes or request by cores per block. When this is set to false, nodes_per_block is computed by cores_per_block / cores_per_node. Default is True.

```
__init__(channel=LocalChannel(envs={}, script_dir=None,
    userhome='/home/docs/checkouts/readthedocs.org/user_builds/parsl/checkouts/latest/docs'),
    nodes_per_block=1, cores_per_block=None, cores_per_node=None, init_blocks=1,
    min_blocks=0, max_blocks=1, parallelism=1, walltime='00:10:00', scheduler_options="",
    worker_init="", project=None, queue=None, cmd_timeout=120, move_files=True,
    bsub_redirection=False, request_by_nodes=True, launcher=SingleNodeLauncher(debug=True,
    fail_on_any=False))
```

Methods

<code>__init__</code> ([channel, nodes_per_block, ...])	
<code>cancel</code> (job_ids)	Cancels the jobs specified by a list of job ids
<code>execute_wait</code> (cmd[, timeout])	
<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])	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.

`cancel`(job_ids)

Cancels the jobs specified by a list of job ids

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

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

property `status_polling_interval`

Returns the interval, in seconds, at which the status method should be called.

Returns

the number of seconds to wait between calls to status()

`submit`(command, tasks_per_node, job_name='parsl.lsf')

Submit the command as an LSF job.

Parameters

- **command** (*str*) – Command to be made on the remote side.
- **tasks_per_node** (*int*) – Command invocations to be launched per node
- **job_name** (*str*) – Name for the job (must be unique).

Returns

If at capacity, returns None; otherwise, a string identifier for the job

Return type

None or *str*

parsl.providers.SlurmProvider

```
class parsl.providers.SlurmProvider(partition: str | None = None, account: str | None = None, qos: str |
None = None, constraint: str | None = None, channel: Channel =
LocalChannel(envs={}, script_dir=None, user-
home='/home/docs/checkouts/readthedocs.org/user_builds/parsl/checkouts/latest/docs'),
nodes_per_block: int = 1, cores_per_node: int | None = None,
mem_per_node: int | None = None, init_blocks: int = 1, min_blocks:
int = 0, max_blocks: int = 1, parallelism: float = 1, walltime: str =
'00:10:00', scheduler_options: str = "", regex_job_id: str = 'Submitted
batch job (?P<id>\S*)', worker_init: str = "", cmd_timeout: int = 10,
exclusive: bool = True, move_files: bool = True, launcher: Launcher
= SingleNodeLauncher(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 unspecified or None, no partition slurm directive will be specified.
- **account** (*str*) – Slurm account to which to charge resources used by the job. If unspecified or None, the job will use the user's default account.
- **qos** (*str*) – Slurm queue to place job in. If unspecified or None, no queue slurm directive will be specified.
- **constraint** (*str*) – Slurm job constraint, often used to choose cpu or gpu type. If unspecified or None, no constraint slurm directive will be added.
- **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.
- **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.
- **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.
- **regex_job_id** (*str*) – The regular expression used to extract the job ID from the sbatch standard output. The default is `r"Submitted batch job (?P<id>\S*)"`, where `id` is the regular expression symbolic group for the job ID.
- **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: str | None = None, account: str | None = None, qos: str | None = None, constraint: str | None = None, channel: Channel = LocalChannel(envs={}, script_dir=None, userhome='/home/docs/checkouts/readthedocs.org/user_builds/parsl/checkouts/latest/docs'), nodes_per_block: int = 1, cores_per_node: int | None = None, mem_per_node: int | None = None, init_blocks: int = 1, min_blocks: int = 0, max_blocks: int = 1, parallelism: float = 1, walltime: str = '00:10:00', scheduler_options: str = "", regex_job_id: str = 'Submitted batch job (?P<id>\S*)', worker_init: str = "", cmd_timeout: int = 10, exclusive: bool = True, move_files: bool = True, launcher: Launcher = SingleNodeLauncher(debug=True, fail_on_any=False))
```

Methods

<code>__init__</code> ([partition, account, qos, ...])	
<code>cancel</code> (job_ids)	Cancels the jobs specified by a list of job ids
<code>execute_wait</code> (cmd[, timeout])	
<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])	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.

`cancel`(job_ids)

Cancels the jobs specified by a list of job ids

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

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

property status_polling_interval

Returns the interval, in seconds, at which the status method should be called.

Returns

the number of seconds to wait between calls to status()

submit(*command*: *str*, *tasks_per_node*: *int*, *job_name*='parsl.slurm') → *str*

Submit the command as a slurm job.

Parameters

- **command** (*str*) – Command to be made on the remote side.
- **tasks_per_node** (*int*) – Command invocations to be launched per node
- **job_name** (*str*) – Name for the job

Returns

job id – A string identifier for the job

Return type

str

parsl.providers.TorqueProvider

```
class parsl.providers.TorqueProvider(channel=LocalChannel(envs={}, script_dir=None, user-
home='/home/docs/checkouts/readthedocs.org/user_builds/parsl/checkouts/latest/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 qsub to submit, qstat for status, and qdel to cancel jobs. The qsub 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,
userhome='/home/docs/checkouts/readthedocs.org/user_builds/parsl/checkouts/latest/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)
```

Methods

<code>__init__</code> ([channel, account, queue, ...])	
<code>cancel</code> (job_ids)	Cancels the jobs specified by a list of job ids
<code>execute_wait</code> (cmd[, timeout])	
<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])	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.

`cancel`(job_ids)

Cancels the jobs specified by a list of job ids

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

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

property `status_polling_interval`

Returns the interval, in seconds, at which the status method should be called.

Returns

the number of seconds to wait between calls to status()

`submit`(command, tasks_per_node, job_name='parsl.torque')

Submits the command onto an Local Resource Manager job. Submit returns an ID that corresponds to the task that was just submitted.

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

If `tasks_per_node == 1`:

A single node is provisioned

If `tasks_per_node > 1` :

`tasks_per_node` number of nodes are provisioned.

Parameters

- **command** (-) – (String) Commandline invocation to be made on the remote side.
- **tasks_per_node** (-) – command invocations to be launched per node

Kwargs:

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

Returns

At capacity, cannot provision more - `job_id`: (string) Identifier for the job

Return type

- None

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: str | None = None, user_id: str | None = None, group_id: str | None = None, run_as_non_root: bool = False, secret: str | None = 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*) – The Kubernetes ImagePullSecret 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: str | None = None, user_id: str | None = None, group_id: str | None = None, run_as_non_root: bool = False, secret: str | None = None, persistent_volumes: List[Tuple[str, str]] = []*) → None

Methods

<code>__init__</code> (<i>image</i> [, <i>namespace</i> , ...])	
<code>cancel</code> (<i>job_ids</i>)	Cancels the jobs specified by a list of job ids Args: <i>job_ids</i> : [<job_id> ...] Returns : [True/False...] : If the cancel operation fails the entire list will be False.
<code>status</code> (<i>job_ids</i>)	Get the status of a list of jobs identified by the job identifiers returned from the submit request.
<code>submit</code> (<i>cmd_string</i> , <i>tasks_per_node</i> [, <i>job_name</i>])	Submit a job :param - <i>cmd_string</i> : (String) - Name of the container to initiate :param - <i>tasks_per_node</i> : command invocations to be launched per node :type - <i>tasks_per_node</i> : 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.

cancel(*job_ids*)

Cancels the jobs specified by a list of job ids Args: *job_ids* : [<job_id> ...] Returns : [True/False...] : If the cancel operation fails the entire list will be False.

property label

Provides the label for this provider

status(*job_ids*)

Get the status of a list of jobs identified by the job identifiers returned from the submit request. :param - *job_ids*: A list of job identifiers :type - *job_ids*: list

Returns

- A list of JobStatus objects corresponding to each *job_id* in the *job_ids* list.

Raises

- **ExecutionProviderExceptions or its subclasses** -

property status_polling_interval

Returns the interval, in seconds, at which the status method should be called.

Returns

the number of seconds to wait between calls to status()

submit(*cmd_string*, *tasks_per_node*, *job_name*=*'parsl'*)

Submit a job :param - *cmd_string*: (String) - Name of the container to initiate :param - *tasks_per_node*: command invocations to be launched per node :type - *tasks_per_node*: int

Kwargs:

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

Returns

At capacity, cannot provision more - *job_id*: (string) Identifier for the job

Return type

- None

parsl.providers.PBSProProvider

```
class parsl.providers.PBSProProvider(channel=LocalChannel(envs={}, script_dir=None, user_home='/home/docs/checkouts/readthedocs.org/user_builds/parsl/checkouts/latest/docs', account=None, queue=None, scheduler_options="", select_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)
```

PBS Pro Execution Provider

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.
- **select_options** (*str*) – String to append to the #PBS -l select block in the submit script to the scheduler. This can be used to specify ngpus.
- **worker_init** (*str*) – Command to be run before starting a worker, such as ‘module load Anaconda; source activate env’.
- **launcher** (*Launcher*) – Launcher for this provider. The default is [SingleNodeLauncher](#).

```
__init__(channel=LocalChannel(envs={}, script_dir=None,
userhome='/home/docs/checkouts/readthedocs.org/user_builds/parsl/checkouts/latest/docs'),
account=None, queue=None, scheduler_options="", select_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)
```

Methods

<code>__init__</code> ([channel, account, queue, ...])	
<code>cancel</code> (job_ids)	Cancels the jobs specified by a list of job ids
<code>execute_wait</code> (cmd[, timeout])	
<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])	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.

property status_polling_interval

Returns the interval, in seconds, at which the status method should be called.

Returns

the number of seconds to wait between calls to status()

submit(*command*, *tasks_per_node*, *job_name*='parsl')

Submits the command job.

Parameters

- **command** (*str*) – Command to be executed on the remote side.
- **tasks_per_node** (*int*) – Command invocations to be launched per node.
- **job_name** (*str*) – Identifier for job.

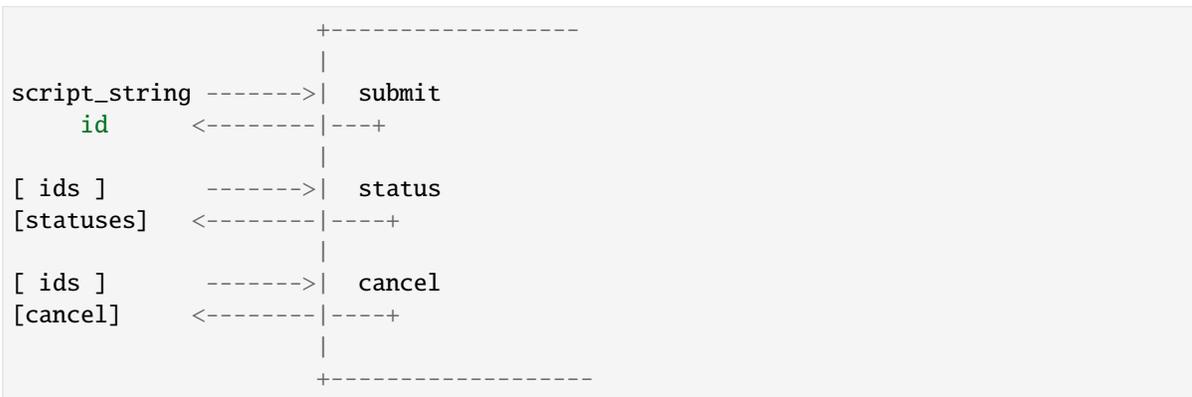
Returns

- *None* – If at capacity and cannot provision more
- **job_id** (*str*) – Identifier for the job

parsl.providers.base.ExecutionProvider**class parsl.providers.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.



abstract `__init__()` → *None*

Methods

<code>__init__()</code>	
<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 for <code>HighThroughputExecutor</code> or <code>WorkQueueExecutor</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.

abstract `cancel(job_ids: List[object]) → List[bool]`

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

Parameters

job_ids (-) – A list of job identifiers

Returns

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

Raises

– **ExecutionProviderException or its subclasses** –

property `cores_per_node: int | None`

Number of cores to provision per node.

Providers which set this property should ask for `cores_per_node` cores when provisioning resources, and set the corresponding environment variable `PARSL_CORES` before executing submitted commands.

If this property is set, executors may use it to calculate how many tasks can run concurrently per node.

abstract property `label: str`

Provides the label for this provider

property `mem_per_node: float | None`

Real memory to provision per node in GB.

Providers which set this property should ask for `mem_per_node` of memory when provisioning resources, and set the corresponding environment variable `PARSL_MEMORY_GB` before executing submitted commands.

If this property is set, executors may use it to calculate how many tasks can run concurrently per node.

abstract status(*job_ids: List[object]*) → List[*JobStatus*]

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

Parameters

job_ids (-) – A list of job identifiers

Returns

- A list of *JobStatus* objects corresponding to each *job_id* in the *job_ids* list.

Raises

– **ExecutionProviderException or its subclasses** –

abstract property status_polling_interval: int

Returns the interval, in seconds, at which the status method should be called.

Returns

the number of seconds to wait between calls to status()

abstract submit(*command: str, tasks_per_node: int, job_name: str = 'parsl.auto'*) → object

The submit method takes the command string to be executed upon instantiation of a resource most often to start a pilot (such as for *HighThroughputExecutor* or *WorkQueueExecutor*).

Args :

- **command** (str) : The bash command string to be executed
- **tasks_per_node** (int) : command invocations to be launched per node

KWargs:

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

Returns

- A job identifier, this could be an integer, string etc or None or any other object that evaluates to boolean false if submission failed but an exception isn't thrown.

Raises

– **ExecutionProviderException or its subclasses** –

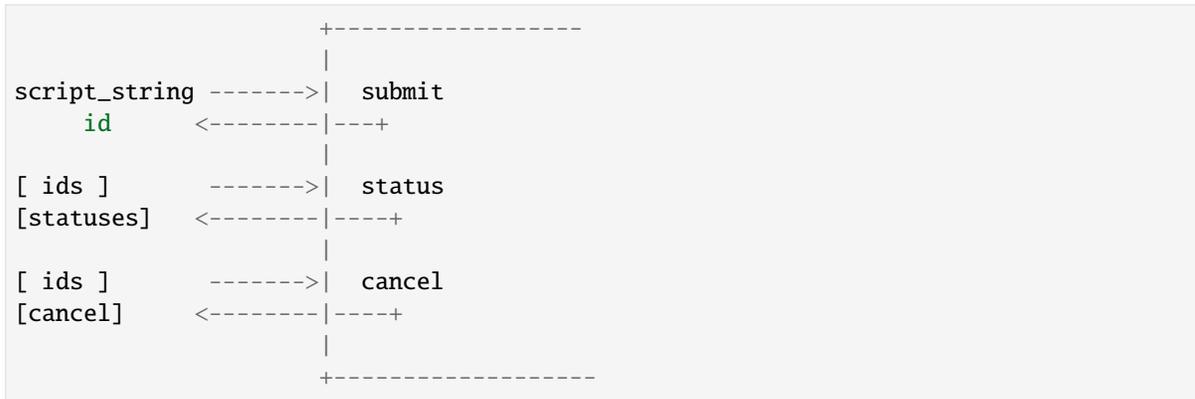
parsl.providers.cluster_provider.ClusterProvider

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** (*Launcher*) – Launcher for this provider.
- **cmd_timeout** (*int*) – Timeout for commands made to the scheduler in seconds



`__init__(label, channel, nodes_per_block, init_blocks, min_blocks, max_blocks, parallelism, walltime, launcher, cmd_timeout=10)`

Methods

<code>__init__(label, channel, nodes_per_block, ...)</code>	
<code>cancel(job_ids)</code>	Cancels the resources identified by the <code>job_ids</code> 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 for <code>HighThroughputExecutor</code> or <code>WorkQueueExecutor</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.

`execute_wait(cmd, timeout=None)`

property label

Provides the label for this provider

status(job_ids)

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

Parameters

`job_ids (-)` – A list of job identifiers

Returns

- A list of JobStatus objects corresponding to each job_id in the job_ids list.

Raises

- **ExecutionProviderException or its subclasses** -

Batch jobs

<code>parsl.jobs.states.JobState</code>	Defines a set of states that a job can be in
<code>parsl.jobs.states.JobStatus</code>	Encapsulates a job state together with other details:
<code>parsl.jobs.error_handlers.noop_error_handler</code>	
<code>parsl.jobs.error_handlers.simple_error_handler</code>	
<code>parsl.jobs.error_handlers.windowed_error_handler</code>	

parsl.jobs.states.JobState

class `parsl.jobs.states.JobState`(*value*, *names=None*, *, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Defines a set of states that a job can be in

`__init__`(*args, **kwds)

Methods

<code>__init__</code> (*args, **kwds)	
<code>as_integer_ratio</code> ()	Return integer ratio.
<code>bit_count</code> ()	Number of ones in the binary representation of the absolute value of self.
<code>bit_length</code> ()	Number of bits necessary to represent self in binary.
<code>conjugate</code>	Returns self, the complex conjugate of any int.
<code>from_bytes</code> ([byteorder, signed])	Return the integer represented by the given array of bytes.
<code>to_bytes</code> ([length, byteorder, signed])	Return an array of bytes representing an integer.

Attributes

<i>UNKNOWN</i>	The batch provider is unable to determinate a state for this job
<i>PENDING</i>	"This job is in the batch queue but has not started running
<i>RUNNING</i>	This job is running in the batch system
<i>CANCELLED</i>	This job has been cancelled
<i>COMPLETED</i>	This job has completed
<i>FAILED</i>	This job has failed
<i>TIMEOUT</i>	This job has ended due to walltime expiry.
<i>HELD</i>	This job is held/suspended in the batch system
<i>MISSING</i>	This job has reached a terminal state without the resources(managers/workers) launched in the job connecting back to the Executor.
denominator	the denominator of a rational number in lowest terms
imag	the imaginary part of a complex number
numerator	the numerator of a rational number in lowest terms
real	the real part of a complex number

CANCELLED = 3

This job has been cancelled

COMPLETED = 4

This job has completed

FAILED = 5

This job has failed

HELD = 7

This job is held/suspended in the batch system

MISSING = 8

This job has reached a terminal state without the resources(managers/workers) launched in the job connecting back to the Executor. This state is set by HTEX when it is able to infer that the block failed to start workers for eg due to bad worker environment or network connectivity issues.

PENDING = 1

"This job is in the batch queue but has not started running

RUNNING = 2

This job is running in the batch system

TIMEOUT = 6

This job has ended due to walltime expiry. This is different from other error states, because in the pilot job model, a timeout is usually expected and not a failure. Timeouts should not be reported with FAILED state: if they are reported as FAILED, Parsl's error handling code in *simple_error_handler* will eventually regard the batch system as broken and shut down.

UNKNOWN = 0

The batch provider is unable to determinate a state for this job

parsl.jobs.states.JobStatus

```
class parsl.jobs.states.JobStatus(state: JobState, message: str | None = None, exit_code: int | None =
                                None, stdout_path: str | None = None, stderr_path: str | None = None)
```

Encapsulates a job state together with other details:

Parameters

- **state** – The machine-readable state of the job this status refers to
- **message** – Optional human readable message
- **exit_code** – Optional exit code
- **stdout_path** – Optional path to a file containing the job’s stdout
- **stderr_path** – Optional path to a file containing the job’s stderr

```
__init__(state: JobState, message: str | None = None, exit_code: int | None = None, stdout_path: str | None
         = None, stderr_path: str | None = None)
```

Methods

```
__init__(state[, message, exit_code, ...])
```

Attributes

```
SUMMARY_TRUNCATION_THRESHOLD
```

```
status_name
```

```
stderr
```

```
stderr_summary
```

```
stdout
```

```
stdout_summary
```

```
terminal
```

```
SUMMARY_TRUNCATION_THRESHOLD = 2048
```

```
property status_name: str
```

```
property stderr: str | None
```

```
property stderr_summary: str | None
```

```
property stdout: str | None
```

property `stdout_summary`: `str` | `None`

property `terminal`: `bool`

`parsl.jobs.error_handlers.noop_error_handler`

`parsl.jobs.error_handlers.noop_error_handler`(*executor*: `BlockProviderExecutor`, *status*: `Dict[str, JobStatus]`, *threshold*: `int = 3`) → `None`

`parsl.jobs.error_handlers.simple_error_handler`

`parsl.jobs.error_handlers.simple_error_handler`(*executor*: `BlockProviderExecutor`, *status*: `Dict[str, JobStatus]`, *threshold*: `int = 3`) → `None`

`parsl.jobs.error_handlers.windowed_error_handler`

`parsl.jobs.error_handlers.windowed_error_handler`(*executor*: `BlockProviderExecutor`, *status*: `Dict[str, JobStatus]`, *threshold*: `int = 3`) → `None`

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 <code>@bash_app</code>
<code>parsl.app.errors.MissingOutputs</code>	Error raised at the end of app execution due to missing output files.
<code>parsl.app.errors.ParslError</code>	Base class for all exceptions.
<code>parsl.errors.ConfigurationError</code>	Raised when a component receives an invalid configuration.
<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 executor related exceptions.
<code>parsl.executors.errors.ScalingFailed</code>	Scaling failed due to error in Execution provider.
<code>parsl.executors.errors.BadMessage</code>	Mangled/Poorly formatted/Unsupported message received
<code>parsl.dataflow.errors.DataFlowException</code>	Base class for all exceptions.
<code>parsl.dataflow.errors.BadCheckpoint</code>	Error raised at the end of app execution due to missing output files.
<code>parsl.dataflow.errors.DependencyError</code>	Error raised if an app cannot run because there was an error
<code>parsl.dataflow.errors.JoinError</code>	Error raised if apps joining into a <code>join_app</code> raise exceptions.

continues on next page

Table 1 – continued from previous page

<code>parsl.launchers.errors.BadLauncher</code>	Error raised when an object of inappropriate type is supplied as a Launcher
<code>parsl.providers.errors.ExecutionProviderException</code>	Base class for all exceptions Only to be invoked when only a more specific error is not available.
<code>parsl.providers.errors.ScaleOutFailed</code>	Scale out failed in the submit phase on the provider side
<code>parsl.providers.errors.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.errors.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
<code>parsl.executors.high_throughput.interchange.ManagerLost</code>	Task lost due to manager loss.
<code>parsl.serialize.errors.DeserializationError</code>	Failure at the deserialization of results/exceptions from remote workers.
<code>parsl.serialize.errors.SerializationError</code>	Failure to serialize task objects.

parsl.app.errors.AppBadFormatting

exception `parsl.app.errors.AppBadFormatting`

An error raised during formatting of a bash function.

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

parsl.app.errors.AppTimeout

exception `parsl.app.errors.AppTimeout`

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

parsl.app.errors.BadStdStreamFile

exception `parsl.app.errors.BadStdStreamFile(reason: str)`

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

Contains:

reason(string)

parsl.app.errors.BashAppNoReturn

exception `parsl.app.errors.BashAppNoReturn(reason: str)`

Bash app returned no string.

Contains: reason(string)

parsl.app.errors.BashExitFailure

exception `parsl.app.errors.BashExitFailure(app_name: str, exitcode: int)`

A non-zero exit code returned from a @bash_app

Contains: app name (str) exitcode (int)

parsl.app.errors.MissingOutputs

exception `parsl.app.errors.MissingOutputs(reason: str, outputs: List[File])`

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

Contains: reason(string) outputs(List of files)

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.

parsl.errors.ConfigurationError

exception `parsl.errors.ConfigurationError`

Raised when a component receives an invalid configuration.

parsl.errors.OptionalModuleMissing

exception `parsl.errors.OptionalModuleMissing`(*module_names: List[str], reason: str*)

Error raised when a required module is missing for a optional/extra component

parsl.executors.errors.ExecutorError

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

Base class for executor related exceptions.

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

parsl.executors.errors.ScalingFailed

exception `parsl.executors.errors.ScalingFailed`(*executor: ParslExecutor, reason: str*)

Scaling failed due to error in Execution provider.

parsl.executors.errors.BadMessage

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

Mangled/Poorly formatted/Unsupported message received

parsl.dataflow.errors.DataFlowException

exception `parsl.dataflow.errors.DataFlowException`

Base class for all exceptions.

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

parsl.dataflow.errors.BadCheckpoint

exception `parsl.dataflow.errors.BadCheckpoint` (*reason: str*)

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

Parameters

reason (-) –

Contains: reason (string) dependent_exceptions

parsl.dataflow.errors.DependencyError

exception `parsl.dataflow.errors.DependencyError` (*dependent_exceptions_tids: Sequence[Tuple[Exception, str]], task_id: int*)

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

Parameters

- **dependent_exceptions_tids** (-) – List of exceptions and identifiers for dependencies which failed. The identifier might be a task ID or the repr of a non-DFK Future.
- **task_id** (-) – Task ID of the task that failed because of the dependency error

parsl.dataflow.errors.JoinError

exception `parsl.dataflow.errors.JoinError` (*dependent_exceptions_tids: Sequence[Tuple[BaseException, str | None]], task_id: int*)

Error raised if apps joining into a join_app raise exceptions. There can be several exceptions (one from each joining app), and JoinError collects them all together.

parsl.launchers.errors.BadLauncher

exception `parsl.launchers.errors.BadLauncher` (*launcher: Launcher*)

Error raised when an object of inappropriate type is supplied as a Launcher

parsl.providers.errors.ExecutionProviderException

exception `parsl.providers.errors.ExecutionProviderException`

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

parsl.providers.errors.ScaleOutFailed

exception `parsl.providers.errors.ScaleOutFailed(provider, reason)`

Scale out failed in the submit phase on the provider side

parsl.providers.errors.SchedulerMissingArgs

exception `parsl.providers.errors.SchedulerMissingArgs(missing_keywords, sitename)`

Error raised when the template used to compose the submit script to the local resource manager is missing required arguments

parsl.providers.errors.ScriptPathError

exception `parsl.providers.errors.ScriptPathError(script_path, reason)`

Error raised when the template used to compose the submit script to the local resource manager is missing required arguments

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.

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)

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)

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)

parsl.channels.errors.FileExists

exception `parsl.channels.errors.FileExists(e: Exception, hostname: str, filename: str | None = 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)

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)

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)

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)

parsl.executors.high_throughput.errors.WorkerLost

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

Exception raised when a worker is lost

parsl.executors.high_throughput.interchange.ManagerLost

exception `parsl.executors.high_throughput.interchange.ManagerLost(manager_id: bytes, hostname: str)`

Task lost due to manager loss. Manager is considered lost when multiple heartbeats have been missed.

parsl.serialize.errors.DeserializationError

exception `parsl.serialize.errors.DeserializationError(reason: str)`

Failure at the deserialization of results/exceptions from remote workers.

parsl.serialize.errors.SerializationError

exception `parsl.serialize.errors.SerializationError`(*fname: str*)

Failure to serialize task objects.

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 AppBase to cover the Python App.
<code>parsl.dataflow.dflow.DataFlowKernel</code>	The DataFlowKernel adds dependency awareness to an existing executor.
<code>parsl.dataflow.memoization.id_for_memo</code>	This should return a byte sequence which identifies the supplied value for memoization purposes: for any two calls of <code>id_for_memo</code> , the byte sequence should be the same when the "same" value is supplied, and different otherwise.
<code>parsl.dataflow.memoization.Memoizer</code>	Memoizer is responsible for ensuring that identical work is not repeated.
<code>parsl.dataflow.states.FINAL_STATES</code>	States from which we will never move to another state, because the job has either definitively completed or failed.
<code>parsl.dataflow.states.States</code>	Enumerates the states a parsl task may be in.
<code>parsl.dataflow.taskrecord.TaskRecord</code>	This stores most information about a Parsl task
<code>parsl.jobs.job_status_poller.JobStatusPoller</code>	
<code>parsl.jobs.strategy.Strategy</code>	Scaling strategy.
<code>parsl.utils.Timer</code>	This class will make a callback periodically, with a period specified by the interval parameter.

parsl.app.app.AppBase

class `parsl.app.app.AppBase`(*func: Callable, data_flow_kernel: DataFlowKernel | None = None, executors: List[str] | Literal['all'] = 'all', cache: bool = False, ignore_for_cache: Sequence[str] | None = 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: Callable, data_flow_kernel: DataFlowKernel | None = None, executors: List[str] | Literal['all'] = 'all', cache: bool = False, ignore_for_cache: Sequence[str] | None = None) → 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` (`str|list`) : Labels of the executors that this app can execute over. Default is 'all'.
- `cache` (`Bool`) : Enable caching of this app ?
- `ignore_for_cache` (`sequence|None`): Names of arguments which will be ignored by the caching mechanism.

Returns

- App object.

Methods

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

`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` (`str|list`) : Labels of the executors that this app can execute over. Default is 'all'.
- `cache` (`Bool`) : Enable caching of this app ?
- `ignore_for_cache` (`sequence|None`): Names of arguments which will be ignored by the caching mechanism.

Returns

- App object.

Methods

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

parsl.app.python.PythonApp

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

Extends AppBase to cover the Python App.

```
__init__(func, data_flow_kernel=None, cache=False, executors='all', ignore_for_cache=None, 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` (str|list) : Labels of the executors that this app can execute over. Default is 'all'.
- `cache` (Bool) : Enable caching of this app ?
- `ignore_for_cache` (sequence|None): Names of arguments which will be ignored by the caching mechanism.

Returns

- App object.

Methods

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

parsl.dataflow.dflow.DataFlowKernel

```
class parsl.dataflow.dflow.DataFlowKernel(config: Config)
```

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:



(continues on next page)

(continued from previous page)

```

App_Fu<-----+-- |           |
                | Dependencies met |
                |           task-----+--> +Submit
                |           Ex_Fu<-----+-- |
    
```

`__init__(config: Config) → None`

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>default_std_autopath(taskrecord, kw)</code>	
<code>handle_app_update(task_record, future)</code>	This function is called as a callback when an AppFuture is in its final state.
<code>handle_exec_update(task_record, 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(task_record, inner_app_future)</code>	in-
<code>launch_if_ready(task_record)</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_record)</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>submit(func, app_args, executors, cache, ...)</code>	Add task to the dataflow system.
<code>update_task_state(task_record, new_state)</code>	Updates a task record state, and recording an appropriate change to task state counters.
<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.
---------------------	--

add_executors(*executors*: Sequence[ParslExecutor]) → None

atexit_cleanup() → None

static check_staging_inhibited(*kwargs*: Dict[str, Any]) → bool

checkpoint(*tasks*: Sequence[TaskRecord] | None = None) → str

Checkpoint the dfk incrementally to a checkpoint file.

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

Kwargs:

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

Note: Checkpointing only works if memoization is enabled

Returns

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

cleanup() → None

DataFlowKernel cleanup.

This involves releasing all resources explicitly.

We call `scale_in` on each of the executors and call `executor.shutdown`.

property config: *Config*

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

Returns

- Config object

default_std_autopath(*taskrecord*, *kw*)

handle_app_update(*task_record*: TaskRecord, *future*: AppFuture) → None

This function is called as a callback when an AppFuture is in its final state.

It will trigger post-app processing such as checkpointing.

Parameters

- **task_record** – Task record
- **future** (*Future*) – The relevant app future (which should be consistent with the task structure ‘app_fu’ entry)

handle_exec_update(*task_record*: TaskRecord, *future*: Future) → None

This function is called only as a callback from an execution attempt reaching a final state (either successfully or failing).

It will launch retries if necessary, and update the task structure.

Parameters

- **task_record** (*dict*) – Task record
- **future** (*Future*) – The future object corresponding to the task which
- **callback** (*makes this*) –

handle_join_update(*task_record*: TaskRecord, *inner_app_future*: AppFuture | None) → None

launch_if_ready(*task_record*: TaskRecord) → None

launch_if_ready will launch the specified task, if it is ready to run (for example, without dependencies, and in pending state).

This should be called by any piece of the DataFlowKernel that thinks a task may have become ready to run.

It is not an error to call launch_if_ready on a task that is not ready to run - launch_if_ready will not incorrectly launch that task.

It is also not an error to call launch_if_ready on a task that has already been launched - launch_if_ready will not re-launch that task.

launch_if_ready is thread safe, so may be called from any thread or callback.

launch_task(*task_record*: TaskRecord) → Future

Handle the actual submission of the task to the executor layer.

Parameters

- task_record** – The task record

Returns

Future that tracks the execution of the submitted function

load_checkpoints(*checkpointDirs*: Sequence[str] | None) → Dict[str, Future]

Load checkpoints from the checkpoint files into a dictionary.

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

Kwargs:

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

Returns

- dict containing, hashed -> future mappings

log_task_states() → None

submit(*func*: Callable, *app_args*: Sequence[Any], *executors*: str | Sequence[str], *cache*: bool, *ignore_for_cache*: Sequence[str] | None, *app_kwargs*: Dict[str, Any], *join*: bool = False) → AppFuture

Add task to the dataflow system.

If the app task has the executors attributes not set (default==’all’) the task will be launched on a randomly selected executor from the list of executors. If the app task specifies a particular set of executors, it will be targeted at the specified executors.

Parameters**func** (-) – A function object**KWargs :**

- `app_args` : Args to the function
- **executors (list or string)**
[List of executors this call could go to.] Default='all'
- `cache (Bool)` : To enable memoization or not
- `ignore_for_cache (sequence)` : List of kwargs to be ignored for memoization/checkpointing
- `app_kwargs (dict)` : Rest of the kwargs to the fn passed as dict.

Returns

(AppFuture) [DataFutures,]

update_task_state(*task_record*: TaskRecord, *new_state*: States) → None

Updates a task record state, and recording an appropriate change to task state counters.

wait_for_current_tasks() → None

Waits for all tasks in the task list to be completed, by waiting for their AppFuture to be completed. This method will not necessarily wait for any tasks added after cleanup has started (such as data stageout?)

wipe_task(*task_id*: int) → NoneRemove task with `task_id` from the internal tasks table**parsl.dataflow.memoization.id_for_memo**`parsl.dataflow.memoization.id_for_memo(obj: object, output_ref: bool = False) → bytes``parsl.dataflow.memoization.id_for_memo(obj: object, output_ref: bool = False) → bytes``parsl.dataflow.memoization.id_for_memo(denormalized_list: list, output_ref: bool = False) → bytes``parsl.dataflow.memoization.id_for_memo(denormalized_tuple: tuple, output_ref: bool = False) → bytes``parsl.dataflow.memoization.id_for_memo(denormalized_dict: dict, output_ref: bool = False) → bytes``parsl.dataflow.memoization.id_for_memo(f: function, output_ref: bool = False) → bytes`

This should return a byte sequence which identifies the supplied value for memoization purposes: for any two calls of `id_for_memo`, the byte sequence should be the same when the “same” value is supplied, and different otherwise.

“same” is in quotes about because sameness is not as straightforward as serialising out the content.

For example, for two dicts `x`, `y`:

```
x = {"a":3, "b":4} y = {"b":4, "a":3}
```

then: `x == y`, but their serialization is not equal, and some other functions on `x` and `y` are not equal: `list(x.keys()) != list(y.keys())`

`id_for_memo` is invoked with `output_ref=True` when the parameter is an output reference (a value in the `outputs=[]` parameter of an app invocation).

Memo hashing might be different for such parameters: for example, a user might choose to hash input File content so that changing the content of an input file invalidates memoization. This does not make sense to do for output files: there is no meaningful content stored where an output filename points at memoization time.

parsl.dataflow.memoization.Memoizer

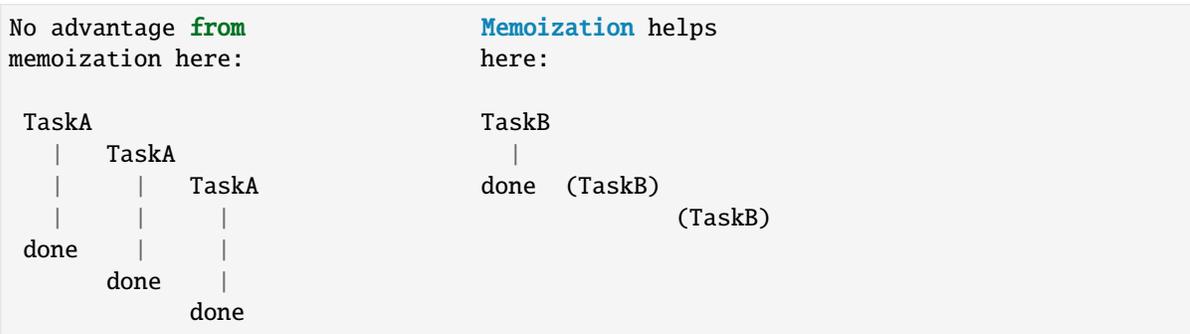
```
class parsl.dataflow.memoization.Memoizer(dfk: DataFlowKernel, memoize: bool = True, checkpoint: Dict[str, Future[Any]] = {})
```

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:



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: DataFlowKernel, memoize: bool = True, checkpoint: Dict[str, Future[Any]] = {})
```

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)</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, r)</code>	Updates the memoization lookup table with the result from a task.

check_memo(*task*: TaskRecord) → Future[Any] | None

Create a hash of the task and its inputs and check the lookup table for this hash.

If present, the results are returned.

Parameters

task (-) – task from the dfk.tasks table

Returns

- Result of the function if present in table, wrapped in a Future

This call will also set task['hashsum'] to the unique hashsum for the func+inputs.

hash_lookup(*hashsum*: str) → Future[Any]

Lookup a hash in the memoization table.

Parameters

hashsum (-) – The same hashes used to uniquely identify apps+inputs

Returns

- Lookup result

Raises

- **KeyError** – if hash not in table

make_hash(*task*: TaskRecord) → str

Create a hash of the task inputs.

Parameters

task (-) – Task dictionary from dfk.tasks

Returns

A unique hash string

Return type

- hash (str)

update_memo(*task*: TaskRecord, *r*: Future[Any]) → None

Updates the memoization lookup table with the result from a task.

Parameters

- **task** (-) – A task dict from dfk.tasks
- **r** (-) – Result future

parsl.dataflow.states.FINAL_STATES

```
parsl.dataflow.states.FINAL_STATES = [<States.exec_done: 3>, <States.memo_done: 9>,
<States.failed: 4>, <States.dep_fail: 5>]
```

States from which we will never move to another state, because the job has either definitively completed or failed.

parsl.dataflow.states.States

```
class parsl.dataflow.states.States(value, names=None, *, module=None, qualname=None, type=None,
start=1, boundary=None)
```

Enumerates the states a parsl task may be in.

These states occur inside the task record for a task inside a *DataFlowKernel* and in the monitoring database.

In a single successful task execution, tasks will progress in this sequence:

```
pending -> launched -> running -> running_ended -> exec_done
```

Other states represent deviations from this path, either due to failures, or to deliberate changes to how tasks are executed (for example due to join_app, or memoization).

All tasks should end up in one of the states listed in *FINAL_STATES*.

```
__init__(*args, **kws)
```

Methods

<code>__init__(*args, **kws)</code>	
<code>as_integer_ratio()</code>	Return integer ratio.
<code>bit_count()</code>	Number of ones in the binary representation of the absolute value of self.
<code>bit_length()</code>	Number of bits necessary to represent self in binary.
<code>conjugate</code>	Returns self, the complex conjugate of any int.
<code>from_bytes([byteorder, signed])</code>	Return the integer represented by the given array of bytes.
<code>to_bytes([length, byteorder, signed])</code>	Return an array of bytes representing an integer.

Attributes

<i>unsched</i>	
<i>pending</i>	Task is known to parsl but cannot run yet.
<i>running</i>	Task is running on a resource.
<i>exec_done</i>	Task has been executed successfully.
<i>failed</i>	Task has failed and no more attempts will be made to run it.
<i>dep_fail</i>	Dependencies of this task failed, so it is marked as failed without even an attempt to launch it.
<i>launched</i>	Task has been passed to a <i>ParslExecutor</i> for execution.
<i>fail_retryable</i>	Task has failed, but can be retried
<i>memo_done</i>	Task was found in the memoization table, so it is marked as done without even an attempt to launch it.
<i>joining</i>	Task is a join_app, joining on internal tasks.
<i>running_ended</i>	Like States.running, this state is also not observed by the DFK, but instead only by monitoring.
<i>denominator</i>	the denominator of a rational number in lowest terms
<i>imag</i>	the imaginary part of a complex number
<i>numerator</i>	the numerator of a rational number in lowest terms
<i>real</i>	the real part of a complex number

dep_fail = 5

Dependencies of this task failed, so it is marked as failed without even an attempt to launch it.

exec_done = 3

Task has been executed successfully.

fail_retryable = 8

Task has failed, but can be retried

failed = 4

Task has failed and no more attempts will be made to run it.

joining = 10

Task is a join_app, joining on internal tasks. The task has run its own Python code, and is now waiting on other tasks before it can make further progress (to a done/failed state).

launched = 7

Task has been passed to a *ParslExecutor* for execution.

memo_done = 9

Task was found in the memoization table, so it is marked as done without even an attempt to launch it.

pending = 0

Task is known to parsl but cannot run yet. Usually, a task cannot run because it is waiting for dependency tasks to complete.

running = 2

Task is running on a resource. This state is special - a DFK task record never goes to States.running state; but the monitoring database may represent a task in this state based on non-DFK information received from monitor_wrapper.

running_ended = 11

Like States.running, this state is also not observed by the DFK, but instead only by monitoring. This state does not record anything about task success or failure, merely that the wrapper ran long enough to record it as finished.

unsched = -1

parsl.dataflow.taskrecord.TaskRecord

class parsl.dataflow.taskrecord.TaskRecord

This stores most information about a Parsl task

`__init__(*args, **kwargs)`

Methods

<code>__init__(*args, **kwargs)</code>	
<code>clear()</code>	
<code>copy()</code>	
<code>fromkeys([value])</code>	Create a new dictionary with keys from iterable and values set to value.
<code>get(key[, default])</code>	Return the value for key if key is in the dictionary, else default.
<code>items()</code>	
<code>keys()</code>	
<code>pop(k[,d])</code>	If the key is not found, return the default if given; otherwise, raise a KeyError.
<code>popitem()</code>	Remove and return a (key, value) pair as a 2-tuple.
<code>setdefault(key[, default])</code>	Insert key with a value of default if key is not in the dictionary.
<code>update([E,]**F)</code>	If E is present and has a .keys() method, then does: for k in E: D[k] = E[k] If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]
<code>values()</code>	

Attributes

<i>dfk</i>	The DataFlowKernel which is managing this task.
<i>func_name</i>	
<i>status</i>	
<i>depends</i>	
<i>app_fu</i>	The Future which was returned to the user when an app was invoked.
<i>exec_fu</i>	When a task has been launched on an executor, stores the Future returned by that executor.
<i>executor</i>	The name of the executor which this task will be/is being/was executed on.
<i>retries_left</i>	
<i>fail_count</i>	
<i>fail_cost</i>	
<i>fail_history</i>	
<i>checkpoint</i>	Should this task be checkpointed?
<i>hashsum</i>	The hash used for checkpointing and memoisation.
<i>task_launch_lock</i>	This lock is used to ensure that task launch only happens once.
<i>func</i>	
<i>fn_hash</i>	
<i>args</i>	
<i>kwargs</i>	
<i>time_invoked</i>	
<i>time_returned</i>	
<i>try_time_launched</i>	
<i>try_time_returned</i>	
<i>memoize</i>	Should this task be memoized?
<i>ignore_for_cache</i>	
<i>from_memo</i>	
<i>id</i>	
<i>try_id</i>	

continues on next page

Table 2 – continued from previous page

<code>resource_specification</code>	Dictionary containing relevant info for a task execution.
<code>join</code>	Is this a join_app?
<code>joins</code>	If this is a join app and the python body has executed, then this contains the Future or list of Futures that the join app will join.
<code>join_lock</code>	Restricts access to end-of-join behavior to ensure that joins only complete once, even if several joining Futures complete close together in time.

app_fu: `AppFuture`

The Future which was returned to the user when an app was invoked.

args: `Sequence[Any]`

checkpoint: `bool`

Should this task be checkpointed?

depends: `List[Future]`

dfk: `dflow.DataFlowKernel`

The DataFlowKernel which is managing this task.

exec_fu: `Future | None`

When a task has been launched on an executor, stores the Future returned by that executor.

executor: `str`

The name of the executor which this task will be/is being/was executed on.

fail_cost: `float`

fail_count: `int`

fail_history: `List[str]`

fn_hash: `str`

from_memo: `bool | None`

func: `Callable`

func_name: `str`

hashsum: `str | None`

The hash used for checkpointing and memoisation. This is not known until at least all relevant dependencies have completed, and will be None before that.

id: `int`

ignore_for_cache: `Sequence[str]`

join: `bool`

Is this a join_app?

join_lock: `threading.Lock`

Restricts access to end-of-join behavior to ensure that joins only complete once, even if several joining Futures complete close together in time.

joins: `None` | `Future` | `List[Future]`

If this is a join app and the python body has executed, then this contains the Future or list of Futures that the join app will join.

kwargs: `Dict[str, Any]`

memoize: `bool`

Should this task be memoized?

resource_specification: `Dict[str, Any]`

Dictionary containing relevant info for a task execution. Includes resources to allocate and execution mode as a given executor permits.

retries_left: `int`

status: `States`

task_launch_lock: `threading.Lock`

This lock is used to ensure that task launch only happens once. A task can be launched by dependencies completing from arbitrary threads, and a race condition would exist when dependencies complete in multiple threads very close together in time, which this lock prevents.

time_invoked: `datetime.datetime` | `None`

time_returned: `datetime.datetime` | `None`

try_id: `int`

try_time_launched: `datetime.datetime` | `None`

try_time_returned: `datetime.datetime` | `None`

`parsl.jobs.job_status_poller.JobStatusPoller`

```
class parsl.jobs.job_status_poller.JobStatusPoller(*, strategy: str | None, max_idletime: float,
                                                    strategy_period: float | int, monitoring:
                                                    MonitoringRadio | None = None)
```

```
__init__(*, strategy: str | None, max_idletime: float, strategy_period: float | int, monitoring:
          MonitoringRadio | None = None) → None
```

Initialize the Timer object. We start the timer thread here

KWargs:

- `interval` (int or float) : number of seconds between callback events
- `name` (str) : a base name to use when naming the started thread

Methods

<code>__init__(*, strategy, max_idletime, ...[, ...])</code>	Initialize the Timer object.
<code>add_executors(executors)</code>	
<code>close([timeout])</code>	Merge the threads and terminate.
<code>make_callback()</code>	Makes the callback and resets the timer.
<code>poll()</code>	

`add_executors(executors: Sequence[BlockProviderExecutor]) → None`

`close(timeout: float | None = None) → None`

Merge the threads and terminate.

`poll() → None`

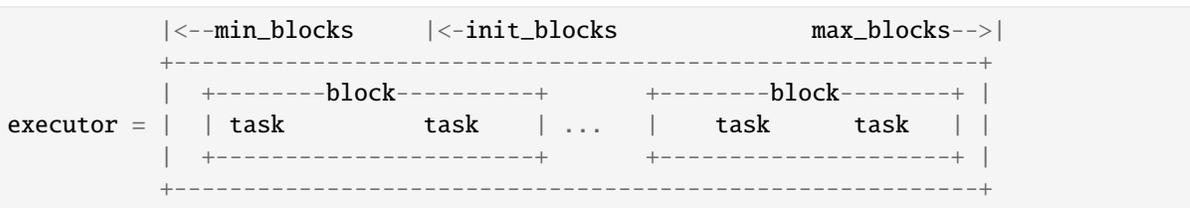
parsl.jobs.strategy.Strategy

`class parsl.jobs.strategy.Strategy(*, strategy: str | None, max_idletime: float)`

Scaling 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__`(* , strategy: str | None, max_idletime: float) → None

Initialize strategy.

Methods

<code>__init__</code> (* , strategy, max_idletime)	Initialize strategy.
<code>add_executors</code> (executors)	

`add_executors`(executors: Sequence[ParslExecutor]) → None

parsl.utils.Timer

class `parsl.utils.Timer`(*callback: Callable, *args: Any, interval: float | int = 5, name: str | None = None*)

This class will make a callback periodically, with a period specified by the interval parameter.

This is based on the following logic :

```
BEGIN (INTERVAL, THRESHOLD, callback) :
    start = current_time()

    while (current_time()-start < INTERVAL) :
        wait()
        break

    callback()
```

__init__(*callback: Callable, *args: Any, interval: float | int = 5, name: str | None = None*) → *None*

Initialize the Timer object. We start the timer thread here

KWargs:

- `interval` (int or float) : number of seconds between callback events
- `name` (str) : a base name to use when naming the started thread

Methods

<code>__init__(callback, *args[, interval, name])</code>	Initialize the Timer object.
<code>close([timeout])</code>	Merge the threads and terminate.
<code>make_callback()</code>	Makes the callback and resets the timer.

close(*timeout: float | None = None*) → *None*

Merge the threads and terminate.

make_callback() → *None*

Makes the callback and resets the timer.

Developer documentation

Contributing

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

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

Roadmap

OVERVIEW

While we follow best practices in software development processes (e.g., CI, flake8, code review), there are opportunities to make our code more maintainable and accessible. This roadmap, written in the fall of 2023, covers our major activities planned through 2025 to increase efficiency, productivity, user experience, and community building.

Features and improvements are documented via GitHub [issues](#) and [pull requests](#).

Code Maintenance

- **Type Annotations and Static Type Checking:** Add static type annotations throughout the codebase and add `typeguard` checks.
- **Release Process:** [Improve the overall release process](#) to synchronize docs and code releases, automatically produce changelog documentation.
- **Components Maturity Model:** Defines the [component maturity model](#) and tags components with their appropriate maturity level.
- **Define and Document Interfaces:** Identify and document interfaces via which [external components](#) can augment the Parsl ecosystem.
- **Distributed Testing Process:** All tests should be run against all possible schedulers, using different executors, on a variety of remote systems. Explore the use of containerized schedulers and remote testing on real systems.

New Features and Integrations

- **Enhanced MPI Support:** Extend Parsl's MPI model with MPI apps and runtime support capable of running MPI apps in different environments (MPI flavor and launcher).
- **Serialization Configuration:** Enable users to select what serialization methods are used and enable users to supply their own serializer.
- **PSI/J integration:** Integrate PSI/J as a common interface for schedulers.
- **Internal Concurrency Model:** Revisit and rearchitect the concurrency model to reduce areas that are not well understood and reduce the likelihood of errors.
- **Common Model for Errors:** Make Parsl errors self-describing and understandable by users.
- **Plug-in Model for External Components:** Extend Parsl to implement interfaces defined above.
- **User Configuration Validation Tool:** Provide tooling to help users configure Parsl and diagnose and resolve errors.
- **Anonymized Usage Tracking:** Usage tracking is crucial for our data-oriented approach to understand the adoption of Parsl, which components are used, and where errors occur. This allows us to prioritize investment in components, progress components through the maturity levels, and identify bugs. Revisit prior usage tracking and develop a service that enables users to control tracking information.
- **Support for Globus Compute:** Enable execution of Parsl tasks using Globus Compute as an executor.
- **Update Globus Data Management:** Update Globus integration to use the new Globus Connect v5 model (i.e., needing specific scopes for individual endpoints).
- **Performance Measurement:** Improve ability to measure performance metrics and report to users.
- **Enhanced Debugging:** Application-level [logging](#) to understand app execution.

Tutorials, Training, and User Support

- **Configuration and Debugging:** Tutorials showing how to configure Parsl for different resources and debug execution.
- **Functional Serialization 101:** Tutorial describing how serialization works and how you can integrate custom serializers.
- **ProxyStore Data Management:** Tutorial showing how you can use ProxyStore to manage data for both inter and intra-site scenarios.
- **Open Dev Calls on Zoom:** The internal core team holds an open dev call/office hours every other Thursday to help users troubleshoot issues, present and share their work, connect with each other, and provide community updates.
- **Project Documentation:** is maintained and updated in [Read the Docs](#).

Longer-term Objectives

- **Globus Compute Integration:** Once Globus Compute supports multi-tenancy, Parsl will be able to use it to run remote tasks on initially one and then later multiple resources.
- **Multi-System Optimization:** Once Globus Compute integration is complete, it is best to use multiple systems for multiple tasks as part of a single workflow.
- **HPC Checkpointing and Job Migration:** As new resources become available, HPC tasks will be able to be checkpointed and moved to the system with more resources.

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

# Make sure to have twine installed
pip3 install twine
```

(continues on next page)

(continued from previous page)

```
# 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/*
```

Doc Docs

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 (```).

Remote builds

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

Local builds

To build the documentation locally, use:

```
$ make clean html
```

To view the freshly rebuilt docs, use:

```
$ cd _build/html
$ python3 -m http.server 8080
```

Once the python http server is launched, point your browser to `http://localhost:8080`

Regenerate module stubs

If necessary, docstring stubs can be regenerated using:

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

Historical Documents

Historical: Changelog

Note: After Parsl 1.2.0, releases moved to a lighter weight automated model. This manual changelog is no longer updated and is now marked as historical.

Change information is delivered as commit messages on individual pull requests, which can be see using any relevant git history browser - for example, on the web at <https://github.com/Parsl/parsl/commits/master/> or on the commandline using using git log.

Parsl 1.2.0

Release date: January 13th, 2022.

Parsl v1.2.0 includes 99 pull requests with contributions from:

Ben Clifford @benclifford, Daniel S. Katz @danielskatz, Douglas Thain @dthain, James Corbett @jameshcorbett, Jonas Rübenach @jrueb, Logan Ward @WardLT, Matthew R. Becker @beckermr, Vladimir @vkhodygo, Yadu Nand Babuji @yadudoc, Yo Yehudi @yochannah, Zhuozhao Li @ZhuozhaoLi, yongyanrao @yongyanrao, Tim Jenness @timj, Darko Marinov @darko-marinov, Quentin Le Boulc'h

High Throughput Executor

- Remove htex self.tasks race condition that shows under high load (#2034)
- Fix htex scale down breakage due to overly aggressive result heartbeat (#2119) [TODO: this fixes a bug introduced since 1.1.0 so note that? #2104]
- Send heartbeats via results connection (#2104)

Work Queue Executor

- Allow use of WorkQueue running_time_min resource constraint (#2113) - WQ recently introduced an additional resource constraint: workers can be aware of their remaining wall time, and tasks can be constrained to only go to workers with sufficient remaining time.
- Implement priority as a Work Queue resource specification (#2067) - The allows a workflow script to influence the order in which queued tasks are executed using Work Queue's existing priority mechanism.
- Disable WQ-level retries with an option to re-enable (#2059) - Previously by default, Work Queue will retry tasks that fail at the WQ level (for example, because of worker failure) an infinite number of times, inside the same parsl-level execution try. That hides the repeated tries from parsl (so monitoring does not report start/end times as might naively be expected for a try, and parsl retry counting does not count).
- Document WorkqueueExecutor project_name remote reporting better (#2089)
- wq executor should show itself using representation mixin (#2064)
- Make WorkQueue worker command configurable (#2036)

Flux Executor

The new FluxExecutor class uses the Flux resource manager (github: flux-framework/flux-core) to launch tasks. Each task is a Flux job.

Condor Provider

- Fix bug in condor provider for unknown jobs (#2161)

LSF Provider

- Update LSF provider to make it more friendly for different LSF-based computers (#2149)

SLURM Provider

- Improve docs and defaults for slurm partition and account parameters. (#2126)

Grid Engine Provider

- missing queue from self - causes config serialisation failure (#2042)

Monitoring

- Index task_hashsum to give cross-run query speedup (#2085)
- Fix monitoring “db locked” errors occurring at scale (#1917)
- Fix worker efficiency plot when tasks are still in progress (#2048)
- Fix use of previously removed reg_time monitoring field (#2020)
- Reorder debug message so it happens when the message is received, without necessarily blocking on the resource_msgs queue put (#2093)

General new features

- Workflow-pluggable retry scoring (#2068) - When a task fails, instead of causing a retry “cost” of 1 (the previous behaviour), this PR allows that cost to be determined by a user specified function which is given some context about the failure.

General bug fixes

- Fix type error when job status output is large. (#2129)
- Fix a race condition in the local channel (#2115)
- Fix incorrect order of manager and interchange versions in error text (#2108)
- Fix to macos multiprocessing spawn and context issues (#2076)
- Tidy tasks_per_node in strategy (#2030)
- Fix and test wrong type handling for joinapp returns (#2063)
- FIX: os independent path (#2043)

Platform and packaging

- Improve support for Windows (#2107)
- Reflect python 3.9 support in setup.py metadata (#2023)
- Remove python <3.6 handling from threadpoolexecutor (#2083)
- Remove breaking .[all] install target (#2069)

Internal tidying

- Remove ipp logging hack in PR #204 (#2170)
- Remove BadRegistration exception definition which has been unused since PR #1671 (#2142)
- Remove AppFuture.__repr__, because superclass Future repr is sufficient (#2143)
- Make monitoring hub exit condition more explicit (#2131)
- Replace parsl's logging NullHandler with python's own NullHandler (#2114)
- Remove a commented out line of dead code in htex (#2116)
- Abstract more block handling from HighThroughputExecutor and share with WorkQueue (#2071)
- Regularise monitoring RESOURCE_INFO messages (#2117)
- Pull os x multiprocessing code into a single module (#2099)
- Describe monitoring protocols better (#2029)
- Remove task_id param from memo functions, as whole task record is available (#2080)
- remove irrelevant __main__ stub of local provider (#2026)
- remove unused weakref_cb (#2022)
- Remove unneeded task_id param from sanitize_and_wrap (#2081)
- Remove outdated IPP related comment in memoization (#2058)
- Remove unused AppBase status field (#2053)
- Do not unwrap joinapp future exceptions unnecessarily (#2084)
- Eliminate self.tasks[id] calls from joinapp callback (#2015)
- Looking at eliminating passing of task IDs and passing task records instead (#2016)

- Eliminate `self.tasks[id]` from `launch_if_ready`
- Eliminate `self.tasks[id]` calls from `launch_task` (#2061)
- Eliminate `self.tasks[id]` from app done callback (#2017)
- Make `process_worker_pool` pass `mypy` (#2052)
- Remove unused `walltime` from `LocalProvider` (#2057)
- Tidy human readable text/variable names around `DependencyError` (#2037)
- Replace old string formatting with f-strings in `utils.py` (#2055)

Documentation, error messages and human-readable text

- Add a documentation chapter summarizing plugin points (#2066)
- Correct docstring for `set_file_logger` (#2156)
- Fix typo in two db error messages and make consistent with each other (#2152)
- Update slack join links to currently unexpired link (#2146)
- small typo fix in doc (#2134)
- Update `CONTRIBUTING.rst` (#2144)
- trying to fix broken link in GitHub (#2133)
- Add `CITATION.cff` file (#2100)
- Refresh the `sanitize_and_wrap` docstring (#2086)
- Rephrase ad-hoc config doc now that `AdHocProvider` (PR #1297) is implemented (#2096)
- Add research notice to readme (#2097)
- Remove untrue claim that `parsl_resource_specification` keys are case insensitive (#2095)
- Use `zsh` compatible install syntax (#2009)
- Remove documentation that interchange is `walltime` aware (#2082)
- Configure sphinx to put in full documentation for each method (#2094)
- autogenerate sphinx stubs rather than requiring manual update each PR (#2087)
- Update docstring for `handle_app_update` (#2079)
- fix a typo (#2024)
- Switch doc verb from `invocated` to `invoked` (#2088)
- Add documentation on meanings of states (#2075)
- Fix summary sentence of `ScaleOutException` (#2021)
- clarify that `max_workers` is per node (#2056)
- Tidy up slurm state comment (#2035)
- Add `nsc` singapore example config (#2003)
- better formatting (#2039)
- Add missing `f` for an f-string (#2062)
- Rework `__repr__` and `__str__` for `OptionalModuleMissing` (#2025)

- Make executor bad state exception log use the exception (#2155)

CI/testing

- Make changes for CI reliability (#2118)
- Make missing worker test cleanup DFK at end (#2153)
- Tidy bash error codes tests. (#2130)
- Upgrade CI to use recent ubuntu, as old version was deprecated (#2111)
- Remove travis config, replaced by GitHub Actions in PR #2078 (#2112)
- Fix CI broken by dependency package changes (#2105)
- Adding github actions for CI (#2078)
- Test combine() pattern in joinapps (#2054)
- Assert that there should be no doc stubs in version control (#2092)
- Add monitoring dependency to local tests (#2074)
- Put viz test in a script (#2019)
- Reduce the size of recursive fibonacci joinapp testing (#2110)
- Remove disabled midway test (#2028)

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, Sohit 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)

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

- `parsl.executors.WorkQueueExecutor` introduced in v0.9.0 is now in beta. `parsl.executors.WorkQueueExecutor` is designed as a drop-in replacement for `parsl.executors.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.
 - `parsl.executors.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: `parsl.launchers.WrappedLauncher` for launching tasks inside containers.
- `parsl.channels.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 `parsl.executors.HighThroughputExecutor`:
 - By default, the `parsl.executors.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](#)

Parsl 0.9.0

Released on October 25th, 2019

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

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

New Functionality

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

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

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

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

- `@python_app` now takes a `walltime` kwarg to limit the task execution time.
- New file staging API `parsl.data_provider.staging.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`

- `parsl.executors.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 `parsl.monitoring.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.
- `parsl.executors.HighThroughputExecutor` is the recommended executor, and `IPyParallelExecutor` is deprecated.
- `parsl.executors.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](#).

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 Tataru @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 `parsl.executors.HighThroughputExecutor` in the configuration section of the userguide.
- Support for OAuth based SSH with `parsl.channels.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](#)

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 `parsl.providers.KubernetesProvider`.
- Scaling recommendations from study on Bluewaters is now available in the userguide.

Parsl 0.7.1

Released on Jan 18th, 2019

New Functionality

- `parsl.executors.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 `parsl.executors.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 `parsl.executors.HighThroughputExecutor` that led to debug logs overflowing channels and terminating blocks of resource is fixed [issue#738](#)

Parsl 0.7.0

Released on Dec 20st, 2018

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

New functionality

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

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

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

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

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

More information on configuring is available in the [Configuration](#) section.

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

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

```

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

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

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

```

More information on configuring is available in the [Configuration](#) section.

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

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

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

- To support resource-based requests for workers and to maintain uniformity across interfaces, `tasks_per_node` is no longer a **provider** option. Instead, the notion of `tasks_per_node` is defined via executor specific options, for eg:
 - `IPyParallelExecutor` provides `workers_per_node`
 - `parsl.executors.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`
- `parsl.launchers.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 `parsl.providers.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](#)
- `parsl.providers.SlurmProvider` + `parsl.launchers.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](#)

Parsl 0.6.1

Released on July 23rd, 2018.

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

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
↔')
```

(continues on next page)

(continued from previous page)

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

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

- 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.

Parsl 0.5.1

Released. May 15th, 2018.

New functionality

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

Bug Fixes

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

Parsl 0.5.2

Released. June 21st, 2018. This is an emergency release addressing [issue#347](#)

Bug Fixes

- Parsl version conflict with libsubmit 0.4.1 [issue#347](#)

Parsl 0.5.0

Released. Apr 16th, 2018.

New functionality

- Support for Globus file transfers [issue#71](#)

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

- **PathLike behavior for Files [issue#174](#)**

– Files behave like strings here :

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

- Automatic checkpointing modes [issue#106](#)

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

- Support for containers with docker [issue#45](#)

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

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

- **Cleaner logging [issue#85](#)**

- Logs are now written by default to `runinfo/RUN_ID/parsl.log`.
- INFO log lines are more readable and compact
- Local configs are now packaged [issue#96](#)

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

Bug Fixes

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

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`.

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](#)

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.

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

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

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.

Historical: Libsubmit Changelog

Note: As of Parsl 0.7.0 the libsubmit repository has been merged into Parsl and nothing more will appear in this changelog.

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.

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](#)

Historical: Swift vs Parsl

Note: This section describes comparisons between Parsl and an earlier workflow system, Swift, as part of a justification for the early prototyping stages of Parsl development. It is no longer relevant for modern Parsl users, but remains of historical interest.

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

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

Parallel Evaluation

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

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

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

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

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

foreach v in array {
    trace(v)
}
```

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

In parsl, a for loop that **launches** tasks has to complete launches before the control may proceed to the next statement. The first for loop has to simply finish iterating, and launching jobs, which should take $\sim \text{length_of_iterable}/1000$ (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.

Mappers

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

There are two choices here :

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

In Swift, the user defines file mappings like this :

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

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

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

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

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

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

Remote-Execution

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

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

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

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

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

Design

Under construction.

Historical: Performance and Scalability

Note: This scalability review summarises results in a paper, Parsl: Pervasive Parallel Programming in Python, which was published in 2019. The results have not been updated since then. For that reason, this section is marked as historical.

Parsl is designed to scale from small to large systems .

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.

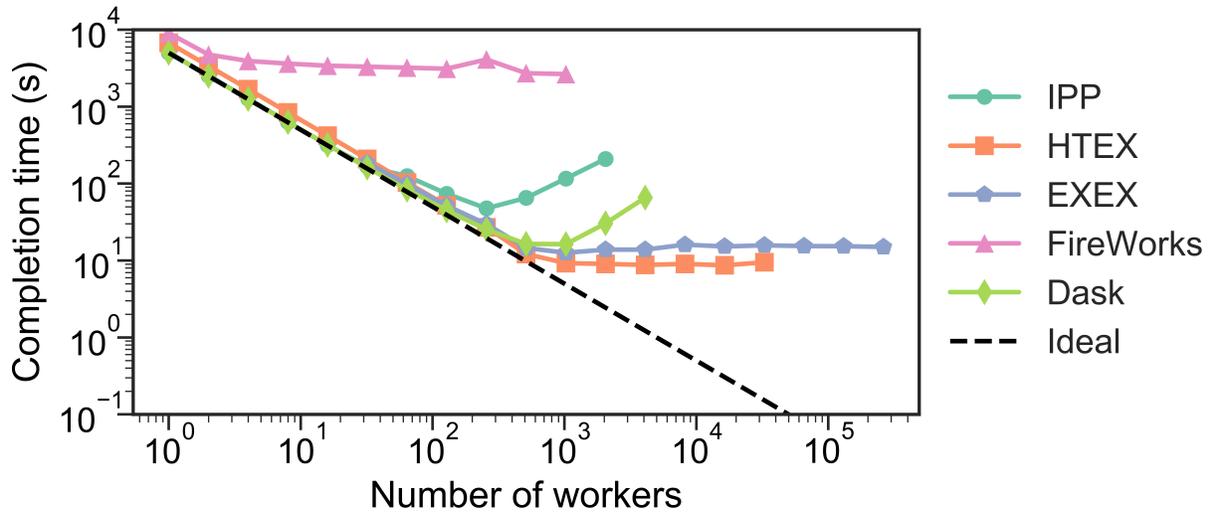
To measure the strong and weak scaling of Parsl executors, we created Parsl programs to run tasks with different durations, ranging from a “no-op”—a Python function that exits immediately—to tasks that sleep for 10, 100, and 1,000 ms. For each executor we deployed a worker per core on each node.

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](#).

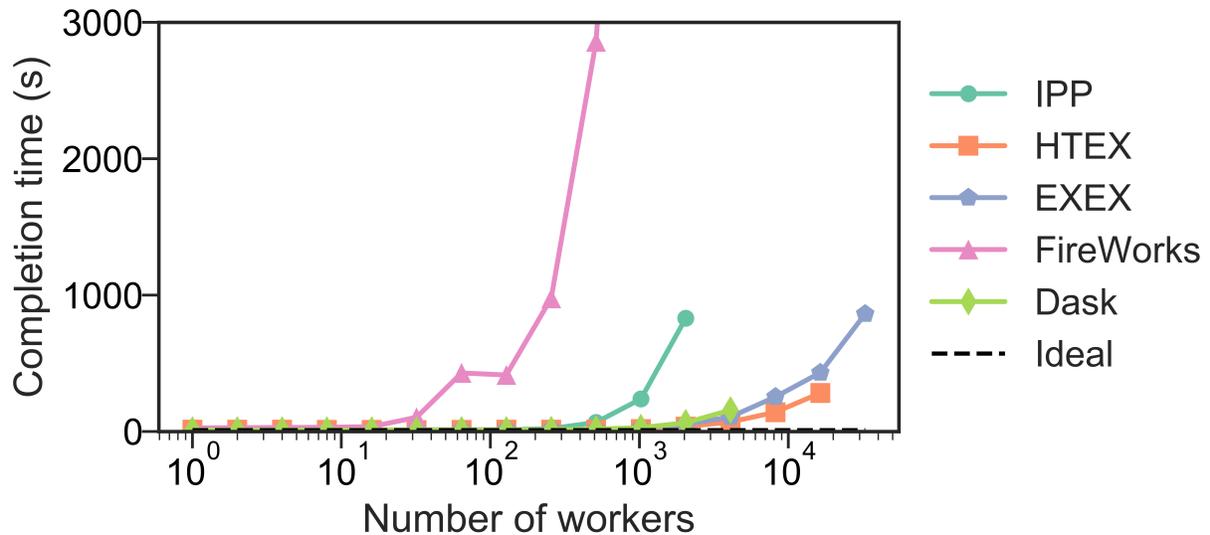
Strong scaling

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 nodes × 32 workers/node × 10 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.



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.

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

INDICES AND TABLES

- genindex
- modindex
- search

Symbols

- `__init__` () (*parsl.app.app.AppBase* method), 221
- `__init__` () (*parsl.app.bash.BashApp* method), 222
- `__init__` () (*parsl.app.futures.DataFuture* method), 133
- `__init__` () (*parsl.app.python.PythonApp* method), 223
- `__init__` () (*parsl.channels.LocalChannel* method), 125
- `__init__` () (*parsl.channels.OAuthSSHChannel* method), 130
- `__init__` () (*parsl.channels.SSHChannel* method), 127
- `__init__` () (*parsl.channels.SSHInteractiveLoginChannel* method), 131
- `__init__` () (*parsl.channels.base.Channel* method), 123
- `__init__` () (*parsl.config.Config* method), 119
- `__init__` () (*parsl.data_provider.data_manager.DataManager* method), 134
- `__init__` () (*parsl.data_provider.file_noop.NoOpFileStaging* method), 141
- `__init__` () (*parsl.data_provider.files.File* method), 137
- `__init__` () (*parsl.data_provider.ftp.FTPInTaskStaging* method), 139
- `__init__` () (*parsl.data_provider.ftp.FTPSeparateTaskStaging* method), 138
- `__init__` () (*parsl.data_provider.globus.GlobusStaging* method), 141
- `__init__` () (*parsl.data_provider.http.HTTPInTaskStaging* method), 144
- `__init__` () (*parsl.data_provider.http.HTTPSeparateTaskStaging* method), 143
- `__init__` () (*parsl.data_provider.rsync.RSyncStaging* method), 145
- `__init__` () (*parsl.data_provider.staging.Staging* method), 136
- `__init__` () (*parsl.dataflow.dflow.DataFlowKernel* method), 224
- `__init__` () (*parsl.dataflow.dflow.DataFlowKernelLoader* method), 116
- `__init__` () (*parsl.dataflow.futures.AppFuture* method), 114
- `__init__` () (*parsl.dataflow.memoization.Memoizer* method), 228
- `__init__` () (*parsl.dataflow.states.States* method), 230
- `__init__` () (*parsl.dataflow.taskrecord.TaskRecord* method), 232
- `__init__` () (*parsl.executors.FluxExecutor* method), 171
- `__init__` () (*parsl.executors.HighThroughputExecutor* method), 156
- `__init__` () (*parsl.executors.ThreadPoolExecutor* method), 153
- `__init__` () (*parsl.executors.WorkQueueExecutor* method), 163
- `__init__` () (*parsl.executors.base.ParsExecutor* method), 147
- `__init__` () (*parsl.executors.radical.RadicalPilotExecutor* method), 173
- `__init__` () (*parsl.executors.status_handling.BlockProviderExecutor* method), 149
- `__init__` () (*parsl.executors.taskvine.TaskVineExecutor* method), 167
- `__init__` () (*parsl.jobs.job_status_poller.JobStatusPoller* method), 235
- `__init__` () (*parsl.jobs.states.JobState* method), 211
- `__init__` () (*parsl.jobs.states.JobStatus* method), 213
- `__init__` () (*parsl.jobs.strategy.Strategy* method), 237
- `__init__` () (*parsl.launchers.AprunLauncher* method), 176
- `__init__` () (*parsl.launchers.GnuParallelLauncher* method), 177
- `__init__` () (*parsl.launchers.JsrunLauncher* method), 178
- `__init__` () (*parsl.launchers.MpiExecLauncher* method), 177
- `__init__` () (*parsl.launchers.SimpleLauncher* method), 175
- `__init__` () (*parsl.launchers.SingleNodeLauncher* method), 175
- `__init__` () (*parsl.launchers.SrunLauncher* method), 176
- `__init__` () (*parsl.launchers.SrunMPILauncher* method), 176
- `__init__` () (*parsl.launchers.WrappedLauncher* method), 178
- `__init__` () (*parsl.launchers.base.Launcher* method), 175
- `__init__` () (*parsl.monitoring.MonitoringHub* method),

117
 __init__() (parsl.providers.AWSProvider method), 182
 __init__() (parsl.providers.AdHocProvider method), 180
 __init__() (parsl.providers.CobaltProvider method), 186
 __init__() (parsl.providers.CondorProvider method), 189
 __init__() (parsl.providers.GoogleCloudProvider method), 191
 __init__() (parsl.providers.GridEngineProvider method), 193
 __init__() (parsl.providers.KubernetesProvider method), 204
 __init__() (parsl.providers.LSFProvider method), 198
 __init__() (parsl.providers.LocalProvider method), 195
 __init__() (parsl.providers.PBSProProvider method), 206
 __init__() (parsl.providers.SlurmProvider method), 200
 __init__() (parsl.providers.TorqueProvider method), 202
 __init__() (parsl.providers.base.ExecutionProvider method), 207
 __init__() (parsl.providers.cluster_provider.ClusterProvider method), 210
 __init__() (parsl.utils.Timer method), 238

A

abspath() (parsl.channels.base.Channel method), 123
 abspath() (parsl.channels.LocalChannel method), 125
 abspath() (parsl.channels.SSHChannel method), 128
 add_executors() (parsl.dataflow.dflow.DataFlowKernel method), 225
 add_executors() (parsl.jobs.job_status_poller.JobStatusPoller method), 236
 add_executors() (parsl.jobs.strategy.Strategy method), 237
 address_by_hostname() (in module parsl.addresses), 121
 address_by_interface() (in module parsl.addresses), 121
 address_by_query() (in module parsl.addresses), 121
 address_by_route() (in module parsl.addresses), 122
 AdHocProvider (class in parsl.providers), 179
 app_fu (parsl.dataflow.taskrecord.TaskRecord attribute), 234
 AppBadFormatting, 215
 AppBase (class in parsl.app.app), 221
 AppException, 216
 AppFuture (class in parsl.dataflow.futures), 114
 AppTimeout, 216
 AprunLauncher (class in parsl.launchers), 176

args (parsl.dataflow.taskrecord.TaskRecord attribute), 234
 atexit_cleanup() (parsl.dataflow.dflow.DataFlowKernel method), 225
 atexit_cleanup() (parsl.executors.taskvine.TaskVineExecutor method), 169
 atexit_cleanup() (parsl.executors.WorkQueueExecutor method), 165
 AuthException, 220
 AWSProvider (class in parsl.providers), 181

B

bad_state_is_set (parsl.executors.status_handling.BlockProviderExecutor property), 151
 BadCheckpoint, 218
 BadHostKeyException, 219
 BadLauncher, 218
 BadMessage, 217
 BadPermsScriptPath, 219
 BadScriptPath, 219
 BadStdStreamFile, 216
 bash_app() (in module parsl.app.app), 113
 BashApp (class in parsl.app.bash), 222
 BashAppNoReturn, 216
 BashExitFailure, 216
 BlockProviderExecutor (class in parsl.executors.status_handling), 149
 bye() (parsl.providers.GoogleCloudProvider method), 191

C

can_stage_in() (parsl.data_provider.file_noop.NoOpFileStaging method), 141
 can_stage_in() (parsl.data_provider.ftp.FTPInTaskStaging method), 140
 can_stage_in() (parsl.data_provider.ftp.FTPSeparateTaskStaging method), 139
 can_stage_in() (parsl.data_provider.globus.GlobusStaging method), 142
 can_stage_in() (parsl.data_provider.http.HTTPInTaskStaging method), 144
 can_stage_in() (parsl.data_provider.http.HTTPSeparateTaskStaging method), 143
 can_stage_in() (parsl.data_provider.rsync.RSyncStaging method), 145
 can_stage_in() (parsl.data_provider.staging.Staging method), 136
 can_stage_out() (parsl.data_provider.file_noop.NoOpFileStaging method), 141
 can_stage_out() (parsl.data_provider.globus.GlobusStaging method), 142
 can_stage_out() (parsl.data_provider.rsync.RSyncStaging method), 145

- `can_stage_out()` (*parsl.data_provider.staging.Staging method*), 136
`cancel()` (*parsl.app.futures.DataFuture method*), 133
`cancel()` (*parsl.dataflow.futures.AppFuture method*), 115
`cancel()` (*parsl.providers.AdHocProvider method*), 180
`cancel()` (*parsl.providers.AWSProvider method*), 183
`cancel()` (*parsl.providers.base.ExecutionProvider method*), 208
`cancel()` (*parsl.providers.CobaltProvider method*), 187
`cancel()` (*parsl.providers.CondorProvider method*), 189
`cancel()` (*parsl.providers.GoogleCloudProvider method*), 191
`cancel()` (*parsl.providers.GridEngineProvider method*), 194
`cancel()` (*parsl.providers.KubernetesProvider method*), 204
`cancel()` (*parsl.providers.LocalProvider method*), 195
`cancel()` (*parsl.providers.LSFProvider method*), 198
`cancel()` (*parsl.providers.SlurmProvider method*), 200
`cancel()` (*parsl.providers.TorqueProvider method*), 202
CANCELLED (*parsl.jobs.states.JobState attribute*), 212
`cancelled()` (*parsl.app.futures.DataFuture method*), 134
`cancelled()` (*parsl.dataflow.futures.AppFuture method*), 115
Channel (*class in parsl.channels.base*), 123
ChannelError, 219
`check_memo()` (*parsl.dataflow.memoization.Memoizer method*), 229
`check_staging_inhibited()` (*parsl.dataflow.dflow.DataFlowKernel static method*), 225
checkpoint (*parsl.dataflow.taskrecord.TaskRecord attribute*), 234
`checkpoint()` (*parsl.dataflow.dflow.DataFlowKernel method*), 225
`cleancopy()` (*parsl.data_provider.files.File method*), 138
`cleanup()` (*parsl.dataflow.dflow.DataFlowKernel method*), 225
`clear()` (*parsl.dataflow.dflow.DataFlowKernelLoader class method*), 116
`close()` (*parsl.channels.base.Channel method*), 123
`close()` (*parsl.channels.LocalChannel method*), 125
`close()` (*parsl.channels.OAuthSSHChannel method*), 131
`close()` (*parsl.channels.SSHChannel method*), 128
`close()` (*parsl.jobs.job_status_poller.JobStatusPoller method*), 236
`close()` (*parsl.monitoring.MonitoringHub method*), 118
`close()` (*parsl.utils.Timer method*), 238
ClusterProvider (*class in parsl.providers.cluster_provider*), 209
CobaltProvider (*class in parsl.providers*), 186
COMPLETED (*parsl.jobs.states.JobState attribute*), 212
CondorProvider (*class in parsl.providers*), 188
Config (*class in parsl.config*), 118
config (*parsl.dataflow.dflow.DataFlowKernel property*), 225
`config_route_table()` (*parsl.providers.AWSProvider method*), 183
ConfigurationError, 217
`connected_blocks()` (*parsl.executors.HighThroughputExecutor method*), 159
`connected_managers()` (*parsl.executors.HighThroughputExecutor method*), 159
`connected_workers` (*parsl.executors.HighThroughputExecutor property*), 159
`cores_per_node` (*parsl.providers.base.ExecutionProvider property*), 208
`create_instance()` (*parsl.providers.GoogleCloudProvider method*), 192
`create_monitoring_info()` (*parsl.executors.status_handling.BlockProviderExecutor method*), 151
`create_name_tag_spec()` (*parsl.providers.AWSProvider method*), 183
`create_session()` (*parsl.providers.AWSProvider method*), 183
`create_vpc()` (*parsl.providers.AWSProvider method*), 184
- ## D
- DataFlowException**, 217
DataFlowKernel (*class in parsl.dataflow.dflow*), 223
DataFlowKernelLoader (*class in parsl.dataflow.dflow*), 116
DataFuture (*class in parsl.app.futures*), 133
DataManager (*class in parsl.data_provider.data_manager*), 134
DEFAULT_LAUNCH_CMD (*parsl.executors.FluxExecutor attribute*), 171
`default_std_autopath()` (*parsl.dataflow.dflow.DataFlowKernel method*), 225
`delete_instance()` (*parsl.providers.GoogleCloudProvider method*), 192
dep_fail (*parsl.dataflow.states.States attribute*), 231
DependencyError, 218
depends (*parsl.dataflow.taskrecord.TaskRecord attribute*), 234
DeserializationError, 220
dfk (*parsl.dataflow.taskrecord.TaskRecord attribute*), 234
`dfk()` (*parsl.dataflow.dflow.DataFlowKernelLoader class method*), 116

E

`exec_done` (*parsl.dataflow.states.States* attribute), 231
`exec_fu` (*parsl.dataflow.taskrecord.TaskRecord* attribute), 234
`execute_wait()` (*parsl.channels.base.Channel* method), 123
`execute_wait()` (*parsl.channels.LocalChannel* method), 126
`execute_wait()` (*parsl.channels.OAuthSSHChannel* method), 131
`execute_wait()` (*parsl.channels.SSHChannel* method), 128
`execute_wait()` (*parsl.providers.cluster_provider.ClusterProvider* method), 210
`ExecutionProvider` (class in *parsl.providers.base*), 207
`ExecutionProviderException`, 218
`executor` (*parsl.dataflow.taskrecord.TaskRecord* attribute), 234
`executor_exception` (*parsl.executors.status_handling.BlockProviderExecutor* property), 151
`ExecutorError`, 217
`executors` (*parsl.config.Config* property), 120

F

`fail_cost` (*parsl.dataflow.taskrecord.TaskRecord* attribute), 234
`fail_count` (*parsl.dataflow.taskrecord.TaskRecord* attribute), 234
`fail_history` (*parsl.dataflow.taskrecord.TaskRecord* attribute), 234
`fail_retryable` (*parsl.dataflow.states.States* attribute), 231
`failed` (*parsl.dataflow.states.States* attribute), 231
`FAILED` (*parsl.jobs.states.JobState* attribute), 212
`File` (class in *parsl.data_provider.files*), 137
`FileCopyException`, 220
`FileExists`, 220
`filename` (*parsl.app.futures.DataFuture* property), 134
`filepath` (*parsl.app.futures.DataFuture* property), 134
`filepath` (*parsl.data_provider.files.File* property), 138
`FINAL_STATES` (in module *parsl.dataflow.states*), 230
`FluxExecutor` (class in *parsl.executors*), 170
`fn_hash` (*parsl.dataflow.taskrecord.TaskRecord* attribute), 234
`from_memo` (*parsl.dataflow.taskrecord.TaskRecord* attribute), 234
`FTPInTaskStaging` (class in *parsl.data_provider.ftp*), 139
`FTPSeparateTaskStaging` (class in *parsl.data_provider.ftp*), 138
`func` (*parsl.dataflow.taskrecord.TaskRecord* attribute), 234
`func_name` (*parsl.dataflow.taskrecord.TaskRecord* attribute), 234

G

`generate_aws_id()` (*parsl.providers.AWSProvider* method), 184
`get_all_checkpoints()` (in module *parsl.utils*), 122
`get_configs()` (*parsl.providers.GridEngineProvider* method), 194
`get_instance_state()` (*parsl.providers.AWSProvider* method), 184
`get_last_checkpoint()` (in module *parsl.utils*), 122
`get_usage_information()` (*parsl.config.Config* method), 120
`get_usage_information()` (*parsl.executors.HighThroughputExecutor* method), 159
`get_zone()` (*parsl.providers.GoogleCloudProvider* method), 192
`GlobusStaging` (class in *parsl.data_provider.globus*), 141
`GridEngineLauncher` (class in *parsl.launchers*), 177
`goodbye()` (*parsl.providers.AWSProvider* method), 184
`GoogleCloudProvider` (class in *parsl.providers*), 190
`GridEngineProvider` (class in *parsl.providers*), 192

H

`handle_app_update()` (*parsl.dataflow.dflow.DataFlowKernel* method), 225
`handle_errors()` (*parsl.executors.status_handling.BlockProviderExecutor* method), 151
`handle_exec_update()` (*parsl.dataflow.dflow.DataFlowKernel* method), 225
`handle_join_update()` (*parsl.dataflow.dflow.DataFlowKernel* method), 226
`hash_lookup()` (*parsl.dataflow.memoization.Memoizer* method), 229
`hashsum` (*parsl.dataflow.taskrecord.TaskRecord* attribute), 234
`HELD` (*parsl.jobs.states.JobState* attribute), 212
`HighThroughputExecutor` (class in *parsl.executors*), 154
`hold_worker()` (*parsl.executors.HighThroughputExecutor* method), 159
`HTTPInTaskStaging` (class in *parsl.data_provider.http*), 144
`HTTPSeparateTaskStaging` (class in *parsl.data_provider.http*), 143
`hub_address` (*parsl.executors.base.ParslExecutor* property), 148
`hub_port` (*parsl.executors.base.ParslExecutor* property), 148

I

`id` (*parsl.dataflow.taskrecord.TaskRecord* attribute), 234
`id_for_memo()` (in module *parsl.dataflow.memoization*), 227
`ignore_for_cache` (*parsl.dataflow.taskrecord.TaskRecord* attribute), 234
`initialize_boto_client()` (*parsl.providers.AWSProvider* method), 184
`initialize_globus()` (*parsl.data_provider.globus.GlobusStaging* method), 142
`initialize_scaling()` (*parsl.executors.HighThroughputExecutor* method), 160
`initialize_scaling()` (*parsl.executors.taskvine.TaskVineExecutor* method), 169
`initialize_scaling()` (*parsl.executors.WorkQueueExecutor* method), 165
`isdir()` (*parsl.channels.base.Channel* method), 124
`isdir()` (*parsl.channels.LocalChannel* method), 126
`isdir()` (*parsl.channels.SSHChannel* method), 129

J

`JobState` (class in *parsl.jobs.states*), 211
`JobStatus` (class in *parsl.jobs.states*), 213
`JobStatusPoller` (class in *parsl.jobs.job_status_poller*), 235
`join` (*parsl.dataflow.taskrecord.TaskRecord* attribute), 234
`join_app()` (in module *parsl.app.app*), 114
`join_lock` (*parsl.dataflow.taskrecord.TaskRecord* attribute), 234
`JoinError`, 218
`joining` (*parsl.dataflow.states.States* attribute), 231
`joins` (*parsl.dataflow.taskrecord.TaskRecord* attribute), 234
`JsrunchLauncher` (class in *parsl.launchers*), 178

K

`KubernetesProvider` (class in *parsl.providers*), 203
`kwargs` (*parsl.dataflow.taskrecord.TaskRecord* attribute), 235

L

`label` (*parsl.executors.base.ParslExecutor* attribute), 148
`label` (*parsl.providers.AdHocProvider* property), 180
`label` (*parsl.providers.AWSProvider* property), 184
`label` (*parsl.providers.base.ExecutionProvider* property), 208
`label` (*parsl.providers.cluster_provider.ClusterProvider* property), 210

`label` (*parsl.providers.KubernetesProvider* property), 205
`label` (*parsl.providers.LocalProvider* property), 196
`launch_if_ready()` (*parsl.dataflow.dflow.DataFlowKernel* method), 226
`launch_task()` (*parsl.dataflow.dflow.DataFlowKernel* method), 226
`launched` (*parsl.dataflow.states.States* attribute), 231
`Launcher` (class in *parsl.launchers.base*), 175
`load()` (*parsl.dataflow.dflow.DataFlowKernelLoader* class method), 116
`load_checkpoints()` (*parsl.dataflow.dflow.DataFlowKernel* method), 226
`LocalChannel` (class in *parsl.channels*), 125
`LocalProvider` (class in *parsl.providers*), 195
`log_task_states()` (*parsl.dataflow.dflow.DataFlowKernel* method), 226
`logdir` (*parsl.executors.HighThroughputExecutor* property), 160
`LSFProvider` (class in *parsl.providers*), 197

M

`make_callback()` (*parsl.utils.Timer* method), 238
`make_hash()` (*parsl.dataflow.memoization.Memoizer* method), 229
`makedirs()` (*parsl.channels.base.Channel* method), 124
`makedirs()` (*parsl.channels.LocalChannel* method), 126
`makedirs()` (*parsl.channels.SSHChannel* method), 129
`ManagerLost`, 220
`max_workers` (*parsl.executors.HighThroughputExecutor* property), 160
`mem_per_node` (*parsl.providers.base.ExecutionProvider* property), 208
`memo_done` (*parsl.dataflow.states.States* attribute), 231
`memoize` (*parsl.dataflow.taskrecord.TaskRecord* attribute), 235
`Memoizer` (class in *parsl.dataflow.memoization*), 228
`MISSING` (*parsl.jobs.states.JobState* attribute), 212
`MissingOutputs`, 216
`monitor_resources()` (*parsl.executors.base.ParslExecutor* method), 148
`monitor_resources()` (*parsl.executors.ThreadPoolExecutor* method), 153
`monitoring_radio` (*parsl.executors.base.ParslExecutor* property), 148
`MonitoringHub` (class in *parsl.monitoring*), 117
`MpiExecLauncher` (class in *parsl.launchers*), 177

N

`noop_error_handler()` (in module *parsl.jobs.error_handlers*), 214

NoOpFileStaging (class in *parsl.data_provider.file_noop*), 141

O

OAuthSSHChannel (class in *parsl.channels*), 130

optionally_stage_in() (parsl.data_provider.data_manager.DataManager method), 135

OptionalModuleMissing, 217

outputs (parsl.dataflow.futures.AppFuture property), 115

outstanding (parsl.executors.HighThroughputExecutor property), 160

outstanding (parsl.executors.status_handling.BlockProviderExecutor property), 151

outstanding (parsl.executors.taskvine.TaskVineExecutor property), 169

outstanding (parsl.executors.WorkQueueExecutor property), 165

P

parent_callback() (parsl.app.futures.DataFuture method), 134

ParslError, 217

ParslExecutor (class in *parsl.executors.base*), 147

PBSProProvider (class in *parsl.providers*), 205

pending (parsl.dataflow.states.States attribute), 231

PENDING (parsl.jobs.states.JobState attribute), 212

poll() (parsl.jobs.job_status_poller.JobStatusPoller method), 236

poll_facade() (parsl.executors.status_handling.BlockProviderExecutor method), 151

prepend_envs() (parsl.channels.SSHChannel method), 129

provider (parsl.executors.status_handling.BlockProviderExecutor property), 151

pull_file() (parsl.channels.base.Channel method), 124

pull_file() (parsl.channels.LocalChannel method), 126

pull_file() (parsl.channels.SSHChannel method), 129

push_file() (parsl.channels.base.Channel method), 124

push_file() (parsl.channels.LocalChannel method), 126

push_file() (parsl.channels.SSHChannel method), 129

python_app() (in module *parsl.app.app*), 113

PythonApp (class in *parsl.app.python*), 223

R

RadicalPilotExecutor (class in *parsl.executors.radical*), 172

radio_mode (parsl.executors.base.ParslExecutor attribute), 148

in radio_mode (parsl.executors.HighThroughputExecutor attribute), 160

radio_mode (parsl.executors.taskvine.TaskVineExecutor attribute), 169

radio_mode (parsl.executors.WorkQueueExecutor attribute), 165

read_state_file() (parsl.providers.AWSProvider method), 184

replace_task() (parsl.data_provider.data_manager.DataManager method), 135

replace_task() (parsl.data_provider.ftp.FTPInTaskStaging method), 140

replace_task() (parsl.data_provider.http.HTTPInTaskStaging method), 144

replace_task() (parsl.data_provider.rsync.RSyncStaging method), 145

replace_task() (parsl.data_provider.staging.Staging method), 136

replace_task_stage_out() (parsl.data_provider.data_manager.DataManager method), 135

replace_task_stage_out() (parsl.data_provider.rsync.RSyncStaging method), 145

replace_task_stage_out() (parsl.data_provider.staging.Staging method), 136

resource_specification (parsl.dataflow.taskrecord.TaskRecord attribute), 235

retries_left (parsl.dataflow.taskrecord.TaskRecord attribute), 235

RSyncStaging (class in *parsl.data_provider.rsync*), 145

run_dir (parsl.executors.base.ParslExecutor property), 148

run_id (parsl.executors.base.ParslExecutor property), 148

running (parsl.dataflow.states.States attribute), 231

RUNNING (parsl.jobs.states.JobState attribute), 212

running() (parsl.app.futures.DataFuture method), 134

running_ended (parsl.dataflow.states.States attribute), 231

S

scale_in() (parsl.executors.HighThroughputExecutor method), 160

scale_in() (parsl.executors.status_handling.BlockProviderExecutor method), 151

scale_in() (parsl.executors.taskvine.TaskVineExecutor method), 169

scale_in() (parsl.executors.WorkQueueExecutor method), 165

scale_in_facade() (parsl.executors.status_handling.BlockProviderExecutor method), 152

- scale_out() (*parsl.executors.status_handling.BlockProviderExecutor* method), 152
- scale_out_facade() (*parsl.executors.status_handling.BlockProviderExecutor* method), 152
- ScaleOutFailed, 219
- ScalingFailed, 217
- SchedulerMissingArgs, 219
- script_dir (*parsl.channels.base.Channel* property), 124
- script_dir (*parsl.channels.LocalChannel* property), 127
- script_dir (*parsl.channels.SSHChannel* property), 129
- ScriptPathError, 219
- security_group() (*parsl.providers.AWSProvider* method), 184
- send() (*parsl.monitoring.MonitoringHub* method), 118
- send_monitoring_info() (*parsl.executors.status_handling.BlockProviderExecutor* method), 152
- SerializationError, 221
- set_bad_state_and_fail_all() (*parsl.executors.status_handling.BlockProviderExecutor* method), 152
- set_file_logger() (in module *parsl*), 121
- set_stream_logger() (in module *parsl*), 120
- show_summary() (*parsl.providers.AWSProvider* method), 184
- shut_down_instance() (*parsl.providers.AWSProvider* method), 184
- shutdown() (*parsl.executors.base.ParslExecutor* method), 148
- shutdown() (*parsl.executors.FluxExecutor* method), 171
- shutdown() (*parsl.executors.HighThroughputExecutor* method), 160
- shutdown() (*parsl.executors.radical.RadicalPilotExecutor* method), 173
- shutdown() (*parsl.executors.taskvine.TaskVineExecutor* method), 169
- shutdown() (*parsl.executors.ThreadPoolExecutor* method), 153
- shutdown() (*parsl.executors.WorkQueueExecutor* method), 165
- simple_error_handler() (in module *parsl.jobs.error_handlers*), 214
- SimpleLauncher (class in *parsl.launchers*), 175
- SingleNodeLauncher (class in *parsl.launchers*), 175
- SlurmProvider (class in *parsl.providers*), 199
- spin_up_instance() (*parsl.providers.AWSProvider* method), 185
- SrunLauncher (class in *parsl.launchers*), 176
- SrunMPILauncher (class in *parsl.launchers*), 176
- SSHChannel (class in *parsl.channels*), 127
- SSHException, 220
- SSHInteractiveLoginChannel (class in *parsl.channels*), 127
- stage_in() (*parsl.data_provider.data_manager.DataManager* method), 135
- stage_in() (*parsl.data_provider.ftp.FTPInTaskStaging* method), 140
- stage_in() (*parsl.data_provider.ftp.FTPSeparateTaskStaging* method), 139
- stage_in() (*parsl.data_provider.globus.GlobusStaging* method), 142
- stage_in() (*parsl.data_provider.http.HTTPInTaskStaging* method), 144
- stage_in() (*parsl.data_provider.http.HTTPSeparateTaskStaging* method), 143
- stage_in() (*parsl.data_provider.rsync.RSyncStaging* method), 146
- stage_in() (*parsl.data_provider.staging.Staging* method), 137
- stage_out() (*parsl.data_provider.data_manager.DataManager* method), 135
- stage_out() (*parsl.data_provider.globus.GlobusStaging* method), 142
- stage_out() (*parsl.data_provider.rsync.RSyncStaging* method), 146
- stage_out() (*parsl.data_provider.staging.Staging* method), 137
- Staging (class in *parsl.data_provider.staging*), 136
- start() (*parsl.executors.base.ParslExecutor* method), 148
- start() (*parsl.executors.FluxExecutor* method), 171
- start() (*parsl.executors.HighThroughputExecutor* method), 160
- start() (*parsl.executors.radical.RadicalPilotExecutor* method), 173
- start() (*parsl.executors.taskvine.TaskVineExecutor* method), 169
- start() (*parsl.executors.ThreadPoolExecutor* method), 153
- start() (*parsl.executors.WorkQueueExecutor* method), 165
- start() (*parsl.monitoring.MonitoringHub* method), 118
- States (class in *parsl.dataflow.states*), 230
- status (*parsl.dataflow.taskrecord.TaskRecord* attribute), 235
- status() (*parsl.executors.HighThroughputExecutor* method), 160
- status() (*parsl.executors.status_handling.BlockProviderExecutor* method), 152
- status() (*parsl.providers.AdHocProvider* method), 180
- status() (*parsl.providers.AWSProvider* method), 185
- status() (*parsl.providers.base.ExecutionProvider* method), 208
- status() (*parsl.providers.cluster_provider.ClusterProvider* method), 210
- status() (*parsl.providers.CondorProvider* method), 180

- 189
 - status() (*parsl.providers.GoogleCloudProvider method*), 192
 - status() (*parsl.providers.KubernetesProvider method*), 205
 - status() (*parsl.providers.LocalProvider method*), 196
 - status_facade (*parsl.executors.status_handling.BlockProviderExecutor property*), 152
 - status_name (*parsl.jobs.states.JobStatus property*), 213
 - status_polling_interval (*parsl.executors.status_handling.BlockProviderExecutor property*), 152
 - status_polling_interval (*parsl.providers.AdHocProvider property*), 180
 - status_polling_interval (*parsl.providers.AWSPProvider property*), 185
 - status_polling_interval (*parsl.providers.base.ExecutionProvider property*), 209
 - status_polling_interval (*parsl.providers.CobaltProvider property*), 187
 - status_polling_interval (*parsl.providers.CondorProvider property*), 190
 - status_polling_interval (*parsl.providers.GoogleCloudProvider property*), 192
 - status_polling_interval (*parsl.providers.GridEngineProvider property*), 194
 - status_polling_interval (*parsl.providers.KubernetesProvider property*), 205
 - status_polling_interval (*parsl.providers.LocalProvider property*), 196
 - status_polling_interval (*parsl.providers.LSFProvider property*), 198
 - status_polling_interval (*parsl.providers.PBSProProvider property*), 206
 - status_polling_interval (*parsl.providers.SlurmProvider property*), 200
 - status_polling_interval (*parsl.providers.TorqueProvider property*), 202
 - stderr (*parsl.dataflow.futures.AppFuture property*), 115
 - stderr (*parsl.jobs.states.JobStatus property*), 213
 - stderr_summary (*parsl.jobs.states.JobStatus property*), 213
 - stdout (*parsl.dataflow.futures.AppFuture property*), 115
 - stdout (*parsl.jobs.states.JobStatus property*), 213
 - stdout_summary (*parsl.jobs.states.JobStatus property*), 213
 - Strategy (*class in parsl.jobs.strategy*), 236
 - submit() (*parsl.dataflow.dflow.DataFlowKernel method*), 226
 - submit() (*parsl.executors.base.ParslExecutor method*), 148
 - submit() (*parsl.executors.FluxExecutor method*), 171
 - submit() (*parsl.executors.HighThroughputExecutor method*), 160
 - submit() (*parsl.executors.radical.RadicalPilotExecutor method*), 173
 - submit() (*parsl.executors.taskvine.TaskVineExecutor method*), 169
 - submit() (*parsl.executors.ThreadPoolExecutor method*), 153
 - submit() (*parsl.executors.WorkQueueExecutor method*), 165
 - submit() (*parsl.providers.AdHocProvider method*), 180
 - submit() (*parsl.providers.AWSPProvider method*), 185
 - submit() (*parsl.providers.base.ExecutionProvider method*), 209
 - submit() (*parsl.providers.CobaltProvider method*), 187
 - submit() (*parsl.providers.CondorProvider method*), 190
 - submit() (*parsl.providers.GoogleCloudProvider method*), 192
 - submit() (*parsl.providers.GridEngineProvider method*), 194
 - submit() (*parsl.providers.KubernetesProvider method*), 205
 - submit() (*parsl.providers.LocalProvider method*), 196
 - submit() (*parsl.providers.LSFProvider method*), 198
 - submit() (*parsl.providers.PBSProProvider method*), 207
 - submit() (*parsl.providers.SlurmProvider method*), 201
 - submit() (*parsl.providers.TorqueProvider method*), 202
 - SUMMARY_TRUNCATION_THRESHOLD (*parsl.jobs.states.JobStatus attribute*), 213
- ## T
- task_launch_lock (*parsl.dataflow.taskrecord.TaskRecord attribute*), 235
 - task_state_cb() (*parsl.executors.radical.RadicalPilotExecutor method*), 174
 - task_status() (*parsl.dataflow.futures.AppFuture method*), 115
 - task_translate() (*parsl.executors.radical.RadicalPilotExecutor method*), 174
 - TaskRecord (*class in parsl.dataflow.taskrecord*), 232

tasks (*parsl.executors.status_handling.BlockProviderExecutor* property), 152

TaskVineExecutor (*class in parsl.executors.taskvine*), 166

teardown() (*parsl.providers.AWSPProvider* method), 185

terminal (*parsl.jobs.states.JobStatus* property), 214

ThreadPoolExecutor (*class in parsl.executors*), 152

tid (*parsl.app.futures.DataFuture* property), 134

tid (*parsl.dataflow.futures.AppFuture* property), 116

time_invoked (*parsl.dataflow.taskrecord.TaskRecord* attribute), 235

time_returned (*parsl.dataflow.taskrecord.TaskRecord* attribute), 235

TIMEOUT (*parsl.jobs.states.JobState* attribute), 212

Timer (*class in parsl.utils*), 238

TorqueProvider (*class in parsl.providers*), 201

try_id (*parsl.dataflow.taskrecord.TaskRecord* attribute), 235

try_time_launched (*parsl.dataflow.taskrecord.TaskRecord* attribute), 235

try_time_returned (*parsl.dataflow.taskrecord.TaskRecord* attribute), 235

workers_per_node (*parsl.executors.WorkQueueExecutor* property), 166

WorkQueueExecutor (*class in parsl.executors*), 161

WrappedLauncher (*class in parsl.launchers*), 178

write_state_file() (*parsl.providers.AWSPProvider* method), 185

X

xstr() (*parsl.providers.AWSPProvider* method), 185

U

UNKNOWN (*parsl.jobs.states.JobState* attribute), 212

unsched (*parsl.dataflow.states.States* attribute), 232

unwrap() (*parsl.executors.radical.RadicalPilotExecutor* method), 174

update_memo() (*parsl.dataflow.memoization.Memoizer* method), 229

update_task_state() (*parsl.dataflow.dflow.DataFlowKernel* method), 227

W

wait_for_current_tasks() (*parsl.dataflow.dflow.DataFlowKernel* method), 227

wait_for_current_tasks() (*parsl.dataflow.dflow.DataFlowKernelLoader* class method), 117

windowed_error_handler() (*in parsl.jobs.error_handlers*), 214

wipe_task() (*parsl.dataflow.dflow.DataFlowKernel* method), 227

worker_logdir (*parsl.executors.HighThroughputExecutor* property), 161

WorkerLost, 220

workers_per_node (*parsl.executors.HighThroughputExecutor* property), 161

workers_per_node (*parsl.executors.status_handling.BlockProviderExecutor* property), 152

workers_per_node (*parsl.executors.taskvine.TaskVineExecutor* property), 170