Parsl Guts

Release 2024.09.04

Ben Clifford

Sep 25, 2024

CONTENTS

1 Introduction 9

2 A sample task execution path 11
2.1 Defining a python_app 12
2.2 Invoking a python_app 13
2.3 The Data Flow Kernel 15
2.4 HighThroughputExecutor.submit() 16
2.5 The Interchange 18
2.6 The Process Worker Pool 19

3 Blocks 23
3.1 Pilot jobs 24
3.2 Starting a block of workers 25
3.3 Launchers 27
3.4 Who starts processes? 27
3.5 Choosing when to start or end a block 28
3.6 block error handling 30
3.7 Worker environments 31

4 Elaborating tasks 33
4.1 Trying tasks many times or not at all 33
4.2 Modifying the arguments to a task 38
4.3 Wrapping tasks with more Python 43
4.4 Join apps: dependencies at the end of a task 45

5 **Understanding the monitoring database** 49
- 5.1 Turning on monitoring . . . 49
- 5.2 Using monitoring information . . . 50
- 5.3 What is stored in the database? . . . 52

6 **Serializing tasks and results with Pickle** 55
- 6.1 Tiny pickle tutorial . . . 57
- 6.2 Functions . . . 57
- 6.3 Exceptions . . . 60
- 6.4 Some objects don't make sense to send to other places 61

7 **Modularity and Plugins** 63
- 7.1 Motivation . . . 63
- 7.2 An example: providers . . . 65
- 7.3 An example: retry policies . . . 65
- 7.4 All the plugin points I can think of . . . 66

8 **Colophon** 71

Index 73

Todo: if try 0 fails *OR IF THERES A SUBMIT ERROR?*

(The original entry (page 34) is located in /home/benc/parsl/src/parslguts/elaborating.rst, line 24.)

Todo: do I want to talk about how parameters are keyed here? YES Note on ignore_for_cache and on plugins (forward ref. plugins)

(The original entry (page 37) is located in /home/benc/parsl/src/parslguts/elaborating.rst, line 117.)

Todo: make a forward reference to *Serializing tasks and results with Pickle* (page 55) section about storing the result (but not the args)

(The original entry (page 37) is located in /home/benc/parsl/src/parslguts/elaborating.rst, line 119.)

Todo: task identity and dependencies: there is a notion of "identity" of a task across runs here, that is different from the inside-a-run identity (aka the task id integer allocated sequentially) – it's the hash of all arguments to the app. So what might look like two different invocations fut1 = a(1); fut2 = a(1) to most of Parsl, is actually two invocations of "the same" task as far as checkpointing is concerned (because the two invocations of `a` have the same argument). Another subtlety here is that this identity can't be computed (and so we can't do any checkpoint-replacement) until the dependencies of a task have been completed - we have to run the dependencies of a task T (perhaps themselves by checkpoint restore) before we can ask if task T itself has been checkpointed.

(The original entry (page 38) is located in /home/benc/parsl/src/parslguts/elaborating.rst, line 121.)

Todo: maybe a simple DAG to modify here based on previous staging talks

---

(The original entry (page 42) is located in /home/benc/parsl/src/parslguts/elaborating.rst, line 221.)

Todo: note about app future completing as soon as the value is available and not waiting till stage-out has happened - See issue #1279.

(The original entry (page 43) is located in /home/benc/parsl/src/parslguts/elaborating.rst, line 226.)

Todo: including rich dependency resolving - but that should be an onwards mention of plugin points? and a note about this being a common mistake. but complicated to implement because it needs to traverse arbitrary structures. which might give a bit of a tie-in to how `id_for_memo` works)

(The original entry (page 43) is located in /home/benc/parsl/src/parslguts/elaborating.rst, line 233.)

Todo: earlier on there should be a state graph. then here the same graph with the joining state.

(The original entry (page 47) is located in /home/benc/parsl/src/parslguts/elaborating.rst, line 321.)

Todo: Summarise by me pointing out that in my mind (not necessarily in the architecture of Parsl) that from a core perspective these are all quite similar, even though the user effects are all very different. Which is a nice way to have an abstraction. And maybe that's an

interesting forwards architecture for Parsl one day…

(The original entry (page 47) is located in /home/benc/parsl/src/parslguts/elaborating.rst, line 331.)

Todo: some visualizations for pieces of this could be loosely disassembled pickle bytecode - otherwise lacking in code-level visualization

(The original entry (page 55) is located in /home/benc/parsl/src/parslguts/pickle.rst, line 4.)

Todo: the "function is in __main__ which is different remotely"

(The original entry (page 58) is located in /home/benc/parsl/src/parslguts/pickle.rst, line 65.)

Todo: f does not have a name

> This can happen in a few ways: the biggest one for Parsl is that a python-app decorated function (yes, that's every app defined using a decorator) - the function body won't be the same as the value assigned to the app name variable. because that vairable is used for the PythonApp object, not the underlying function.
>
> That can be worked around by letting a function get a global name, using a variant of the decorator syntax I talked about n the first chapter:

```
def myfunc(a,b):
  return a+b

myapp = python_app(myfunc)
```

now the underlying function is available with `from wherever import myfunc` and the Parsl app equivalent can be invoked with `myapp(3,4)`.

Another situation where a function does not have a global name is when it is defined as a closure inside another function:

```
def add_const(n):
  def myfunc(a,n):
    return a+n

myapp = python_app(add_const(7))
```

This is pretty common in certain functional styles of Python programming. One way to think about how it is a problem is to try to write an `import` statement to import the underlying function for `myapp`.

(The original entry (page 58) is located in /home/benc/parsl/src/parslguts/pickle.rst, line 67.)

Todo: URL for Python bytecode/virtual machine documentation?

(The original entry (page 59) is located in /home/benc/parsl/src/parslguts/pickle.rst, line 107.)

Todo: backref/crossref the worker environment section - it could point here as justification/understanding of which packages should be installed.

(The original entry (page 60) is located in /home/benc/parsl/src/parslguts/pickle.rst, line 109.)

Todo: also mention cloudpickle as a dill-like pickle extension. They are both installable alongside each other… and people mostly haven't

given me decent argumetns for cloudpickle because people don't dig much into understanding whats going on.

(The original entry (page 60) is located in /home/benc/parsl/src/parslguts/pickle.rst, line 122.)

Todo: i think there's a funcx approach to this that i could link to that turns exceptions into strings, which are basic pickle data types we should always be able to unpickle. see issue #3474. You lose the ability to catch specific exceptions (at least in the standard Python way).

(The original entry (page 61) is located in /home/benc/parsl/src/parslguts/pickle.rst, line 136.)

Todo: one example of plotting

(The original entry (page 52) is located in /home/benc/parsl/src/parslguts/monitoring.rst, line 117.)

Todo: deeper dive into workflow/tasks/try table schema - not trying to be comprehensive of all schemas here but those three are a good set to deal with

---

(The original entry (page 52) is located in /home/benc/parsl/src/parslguts/monitoring.rst, line 124.)

Todo: the core task-related tables can get a hierarchical diagram workflow/task/try+state/resource

---

(The original entry (page 53) is located in /home/benc/parsl/src/parslguts/monitoring.rst, line 132.)

Todo: label the various TaskRecord state transitions (there are only a few relevant here) throughout this doc - it will play nicely with the monitoring DB chapter later, to they are reflected not only in the log but also in the monitoring database.

(The original entry (page 21) is located in /home/benc/parsl/src/parslguts/taskpath.rst, line 268.)

Todo: for each, a sentence or two, and a source code reference

(The original entry (page 66) is located in /home/benc/parsl/src/parslguts/plugins.rst, line 68.)

Todo: ref back to *Elaborating tasks* (page 33) if I write that section

(The original entry (page 69) is located in /home/benc/parsl/src/parslguts/plugins.rst, line 139.)

Todo: link to serialization interface, and to pickle documentation for pickle extensibility

(The original entry (page 69) is located in /home/benc/parsl/src/parslguts/plugins.rst, line 143.)

Todo: source code

(The original entry (page 25) is located in /home/benc/parsl/src/parslguts/blocks.rst, line 61.)

Todo: source code

(The original entry (page 26) is located in /home/benc/parsl/src/parslguts/blocks.rst, line 65.)

Todo: line numbers / source code link

(The original entry (page 26) is located in /home/benc/parsl/src/parslguts/blocks.rst, line 69.)

Todo: a paragraph that in traditional HPC workloads, this launcher command is often responsible for starting multiple copies of your code on the same node - so if you wanted 24 cores used for an MPI code, you might use mpirun (TODO: processes_per_node param) to start 24 copies which would run in parallel. This is not how things work with parsl block workers: both the process worker pool and the WQ/TV equivalents usually manage all the tasks on a node from a single worker. So if you're feeling the temptation to make your launcher launch multiple copies of the pilot job worker, maybe there's something else going wrong? and note this is a common problem in modern times, also with OMP, where multiple layers of software think *they* are the one to spawn multiple processes/threads which leads to exponential explosion of threads. which doesn't necessarily kill your workfload but can lead to myterious performance problems. - also this section should consider *user apps* which make the same assumption (so easily 3 layers to draw diagrams about!)

(The original entry (page 27) is located in /home/benc/parsl/src/parslguts/blocks.rst, line 95.)

Todo: above processes_per_node param

(The original entry (page 28) is located in /home/benc/parsl/src/parslguts/blocks.rst, line 97.)

Todo: reference job status poller

(The original entry (page 28) is located in /home/benc/parsl/src/parslguts/blocks.rst, line 106.)

Todo: reference block draining problem and matthew's work.

(The original entry (page 30) is located in /home/benc/parsl/src/parslguts/blocks.rst, line 147.)

Todo: write error handling section (as two parts of the same feedback loop)

(The original entry (page 30) is located in /home/benc/parsl/src/parslguts/blocks.rst, line 161.)

CHAPTER ONE

INTRODUCTION

Hello.

These are notes for a 3 hour course (6 x 25 minute sessions) for experienced Parsl users who want to level-up their ability to use Parsl or to hack on the Parsl codebase by learning more about how Parsl works inside.

This text is not intended to be a comprehensive guide to all parts of the Parsl codebase, but it should get you started on further exploration.

You might like to have a throw-away installation of Parsl 2024.09.02 that you can work with as you go through these notes.

CHAPTER TWO

A SAMPLE TASK EXECUTION PATH

In this section, I'll walk through the code as it executes a single Parsl task: from defining and invoking an app, through to running on a High Throughput Executor worker, and back again.

Here's a simple workflow that you can run on a single computer. I'll point out as we go which bits would run on a worker node when running on an HPC system - but as far as this section is concerned there isn't much difference between running locally on your laptop vs using a multiple-node configuration.

```
import parsl

def fresh_config():
  return parsl.Config(
    executors=[parsl.HighThroughputExecutor()],
  )

@parsl.python_app
def add(x: int, y: int) -> int:
  return x+y

with parsl.load(fresh_config()):
  print(add(5,3).result())
```

This is nothing fancy: there's a config in my preferred style, with al-

most every parameter using a default value. All that is explicit is to use the High Throughput Executor, rather than the default (and boring) Thread Pool Executor.

I'm going to ignore quite a lot: what happens with parsl.load() and what happens at shutdown; I'm going to defer batch system interactions to the *blocks chapter* (page 23), and this example avoids many of Parsl's workflow features which I will cover in the *task elaboration chapter* (page 33).

I'm going to call the Unix/Python process where this code runs, the *user workflow process*. There will be quite a lot of other processes involved, which I will cover as needed.

2.1 Defining a `python_app`

```
@parsl.python_app
def add(x: int, y: int) -> int:
  return x+y
```

Normally `def` defines a function (or a method) in Python. With the `python_app` decorator (defined at parsl/app/app.py line 108), Parsl makes `def` mean something else: this now defines a *Python app* which mostly looks like a function but with fancy Parsl things added. The relevant fancy thing for this example is that `add` will return a `Future[int]`, a `Future` that will eventually get an `int` inside it, instead of directly returning an int.

This decorator syntax is roughly equivalent to writing this. The example script should behave the same if you substitute this code for the above definition:

```
def add(x: int, y: int) -> int:
  return x+y

add = parsl.python_app(add)
```

What happens is first a regular function called add is defined, so the top level Python symbol add refers initially to that function.

Then the add symbol is redefined, to be the output of calling parsl.python_app with the original add definition as an argument.

parsl.python_app is just a regular function. It's allowed to do anything it wants. At the end, add will end up as whatever that function returns.

What it actually does is replace add with a PythonApp object that wraps the original app function. In the next section, I'll dig into that PythonApp object a bit more.

Looking at types:

A normal function in Python has this type:

```
>>> def somefunc():
>>>   return 7

>>> print(type(somefunc))
<class 'function'>
```

but add looks like this:

```
>>> print(type(add))
<class 'parsl.app.python.PythonApp'>
```

See also:

You can read more about decorators in the Python glossary.

2.2 Invoking a python_app

If add isn't a function, what does this code (that looks like a function invocation) mean?

```
add(5,3)
```

In Python, any class can be used with function call syntax, if it has a `__call__` magic method. Here is the PythonApp implementation, in parsl/app/python.py, line 50 onwards:

```
50  def __call__(self, *args, **kwargs):
51
52  # ...
```

```
77    app_fut = dfk.submit(func, app_args=args,
78                         executors=self.executors,
79                         cache=self.cache,
80                         ignore_for_cache=self.ignore_
   ↪for_cache,
81                         app_kwargs=invocation_kwargs,
82                         join=self.join)
83
84    return app_fut
```

The `PythonApp` implementation of `__call__` doesn't do too much: it massages arguments a bit but delegates all the work to the next component along, the Data Flow Kernel referenced by the `dfk` variable. `dfk.submit` returns immediately, without executing anything. It returns an `AppFuture` which will eventually get the final task result, and `PythonApp` returns that to its own caller. This is the future that a user sees when they invoke an app.

The most important parameters to see are the function to execute, stored in `func` and the arguments in `app_args` (a list of positional arguments) and `app_kwargs` (a `dict` of keyword arguments). Those three things are what we will need later on to invoke our function somewhere else, and a lot of the rest of task flow is about moving these around and sometimes changing them.

See also:

Magic methods surrounded by double underscores are the standard

Python way to make arbitrary classes customize standard Python behaviour. The most common one is probably `__repr__` which allows a class to define how it is rendered as a string. There are lots of others documented in the Python data model.

2.3 The Data Flow Kernel

The code above called the `submit` method on a *Data Flow Kernel* (DFK), the core object that manages a live workflow. That call created a *task* inside the DFK. Every app invocation is paired with a task inside the DFK, and the terminology will use those terms fairly interchangeably. There is also usually only one of these DFK objects around at any time, and so often I'll talk about *the* DFK, not *a* DFK.

The DFK follows the God-object antipattern and is a repository for quite a lot of different pieces of functionality in addition to task handling. For example, it is the class which handles start up and shutdown of all the other pieces of Parsl (including block scaling, executors, monitoring, usage tracking and checkpointing). I'm not going to cover any of that here, but be aware when you look through the code that you will see all of that in addition to task handling (it's the longest file in the codebase).

Inside `dfk.submit` (in parsl/dataflow/dflow.py around line 963) two data structures are created: an `AppFuture` and a `TaskRecord`.

The `AppFuture` is the future that the user will get back from app invocation, almost definitely without a result in it yet. It is a thin layer around Python's built-in concurrent.futures.Future class. This is returned from the `submit` method and onwards back to the user immediately. Later on in execution, this is how task completion will be communicated to the submitting user.

The `TaskRecord` (defined in parsl/dataflow/taskrecord.py) contains most of the state for a task.

From the many fields in `TaskRecord`, what we need for now are fields for the app function, positional and keyword arguments to be able to

invoke the app code, and a reference to the `AppFuture` to communicate the result afterwards.

Most of what happens next is task management that I will cover in *Elaborating tasks* (page 33) - things like waiting for dependencies, file staging, checkpointing. In this example, none of that happens and the DFK will go straight to submitting the task to the High Throughput Executor, giving a second future for the task, the *executor future*.

The DFK will use this executor future to do more task management when the executor finishes executing the task.

I'll dig into DFK much more in *Elaborating tasks* (page 33) - for now, I'll just show that the code makes a submit call to the chosen executor (on line 761):

```
760 with self.submitter_lock:
761   exec_fu = executor.submit(function, task_record[
     ↪'resource_specification'], *args, **kwargs)
```

and then adds a callback onto the executor future to run when the task completes (at line 701):

```
701 exec_fu.add_done_callback(partial(self.handle_exec_
     ↪update, task_record))
```

That callback will fire later as the result comes back. This style of callback is used in a few places to drive state changes asynchronously.

2.4 HighThroughputExecutor.submit()

`executor.submit()` above will send the task to the executor I configured, which is an instance of the High ThroughputExecutor. This is the point at which the task would instead go to Work Queue or one of the other executors, if the configuration was different. I'll cover plugin points like this in more depth in *Modularity and Plugins* (page 63).

The High Throughput Executor consists of a bunch of threads and processes distributed across the various nodes you want to execute tasks on.

Inside the user workflow process, the `submit` method packages the task up for execution and sends it on to the *interchange* process.

Inside the user workflow process, the High Throughput Executor `submit` method (parsl/executors/high_throughput/executor.py, line 632 onwards) packages the task up for execution and sends it on to the *interchange* process:

```
666        fut = Future()
667        fut.parsl_executor_task_id = task_id
668        self.tasks[task_id] = fut
669
670        try:
671            fn_buf = pack_res_spec_apply_message(func, args, kwargs,
672                                                 resource_specification=resource_specification,
673                                                 buffer_threshold=1024 * 1024)
674        except TypeError:
675            raise SerializationError(func.__name__)
676
677        msg = {"task_id": task_id, "buffer": fn_buf}
678
679        # Post task to the outgoing queue
680        self.outgoing_q.put(msg)
681
682        # Return the future
683        return fut
```

The steps here are:

- make the executor future
- map it to the task ID so results handling can find it later

- serialize the task definition (that same triple of function, args, keyword args, along with any resource specification) into a byte stream `fn_buf` that is easier to send over the network (see *Serializing tasks and results with Pickle* (page 55) later)
- construct a message for the interchange pairing the task ID with that byte stream sequence
- send that message on the outgoing queue to the interchange
- return the (empty) executor future back to the DFK

Another piece of code will handle getting results back into that executor future later on.

All of the different processes involved in the High Throughput Executor communicate using ZeroMQ (ZMQ). I won't talk about that in much depth, but it's a messaging layer that (in High Throughput Executor) delivers messages over TCP/IP. The `outgoing_q` above is a ZMQ queue for submitting tasks to the interchange.

2.5 The Interchange

The interchange (defined in parsl/executors/high_throughput/interchange.py) runs alongside the user workflow process on the submitting node. It matches up tasks with available workers: it has a queue of tasks, and it has a queue of process worker pool managers which are ready for work.

Whenever it can match a new task (arriving on the outgoing task queue) with a process worker pool that is ready for work, it will send the task onwards to that worker pool. Otherwise, a queue of either ready tasks or ready workers builds up in the interchange.

The matching process so far has been fairly arbitrary but we have been doing some research on better ways to match workers and tasks - I'll talk a little about that later *when talking about scaling in* (page 23).

The interchange has two ZMQ connections per worker pool (one for sending tasks, one for receiving results) and when this task is matched,

the definition will be sent onwards via the relevant per-pool connection.

2.6 The Process Worker Pool

On each worker node on our HPC system, a copy of the process worker pool will be running - *Blocks* (page 23) will talk about how that comes about. In this example workflow, the local system is the only worker node, so there will only be one worker pool. But in a 1000-node run, there would usually be 1000 worker pools, one running on each of those nodes (although other configurations are possible).

These worker pools connect back to the interchange using two network connections each (ZMQ over TCP) - so on the interchange process you'll need 2 fds per node. This is a common limitation to "number of nodes" scalability of Parsl. (see issue #3022 for a proposal to use one network connection per worker pool)

The source code for the process worker pool livces in parsl/executors/high_throughput/process_worker_pool.py.

The worker pool consists of a few closely linked processes:

- The manager process which interfaces to the interchange (this is why you'll see a jumble of references to managers or worker pools in the code: the manager is the externally facing interface to the worker pool)
- Several worker processes - each worker process is a worker. There are a bunch of configuration parameters and heuristics to decide how many workers to run - this happens near the start of the process worker pool process at parsl/executors/high_throughput/process_worker_pool.py line 210. There is one worker per simultaneous task, so usually one per core or one per node (depending on application preference).

The task arrives at the manager, and the manager dispatches it to a free worker. It is possible there isn't a free worker, becuase of the pre-fetch feature which can help in high throughput situations. In that case, the

task will have to wait in another queue - ready to start execution when a worker becomes free, without any more network activity.

The worker then deserialises the byte package that was originally serialized all the way back in the user submit process, giving python objects for the function to run, the positional arguments and the keyword arguments.

At this point, the worker process can invoke the function with those arguments: the worker pool's `execute_task` method handles that at line 593

Now the original function has run! but in a worker that could have been on a different node.

The function execution is probably going to end in two ways: a result or an exception (actually there is a common third way, which is that it kills the unix-level worker process for example by using far too much memory or by a library segfault - or by the batch job containing the worker pool reaching the end of its run time - that is handled, but I'm ignoring that here)

This result needs to be set on the `AppFuture` back in the user workflow process. It flows back over network connections that parallel the submitting side: first back to the interchange, and then to piece of the High Throughput Executor running inside the submit process.

This final part of the High Throughput Executor is less symmetrical: the user workflow script is not necessarily waiting for any results at this point, so the High Throughput Executor runs a second thread to process results, the *result queue thread* implemented by htex._result_queue_worker. This listens for new results and sets the corresponding executor future.

Once the executor future is set, that causes the `handle_exec_done` callback in the Data Flow Kernel to run. Some interesting task handling might happen here (see *Elaborating tasks* (page 33) - things like retry handling) but in this example, nothing interesting happens and the DFK sets the `AppFuture` result.

Setting the `AppFuture` result wakes up the main thread which is sitting

blocked in the `.result()` part of final bit of the workflow:

```
print(add(5,3).result())
```

. . . and the result can be printed.

So now we're at the end of our simple workflow, and we pass out of the parsl context manager. That causes parsl to do various bits of shutdown. and then the user workflow process falls of the bottom and the process ends.

Todo: label the various TaskRecord state transitions (there are only a few relevant here) throughout this doc - it will play nicely with the monitoring DB chapter later, to they are reflected not only in the log but also in the monitoring database.

CHAPTER

THREE

BLOCKS

In *A sample task execution path* (page 11), I assumed that process worker pools magically existed in the right place: on the local machine with the example configuration, but on HPC worker nodes when running a more serious workflow.

The theme of this chapter is: how to get those process worker pools running on the worker nodes.

The configurations for blocks are usually the most non-portable pieces of a Parsl workflow, because they are closely tied to the behaviour of particular HPC machines: this part of the configuration describes what an HPC machine looks like (at least, as much as Parsl needs to know) and so the descriptions will be different for different machines.

Note: So this is one of the most useful areas for admins and users to contribute documentation. For example, the Parsl user guide has a section with configurations for different machines, and ALCF and NERSC both maintain their own Parsl examples.

3.1 Pilot jobs

Not all executors need worker processes - for example the ThreadPoolExecutor runs tasks locally within the main workflow process. But the High Throughput Executor and others use worker processes running (for example) on the worker nodes of an HPC system.

This is known as the *pilot job model*.

These workers (or pilot jobs) don't need to know what work they will perform when they are launched. Once the workers are running they'll get their own work from (in the High Throughput Executor case) the interchange.

Why do things this way when an HPC system already has a system for running jobs (such as Slurm)? Because the overhead on that kind of job can be very big - those systems are targeted more at the scale of "run one job that uses 1000 nodes for 24 hours" but tasks in Parsl might be subsecond: even getting a new Python process started to run that subsecond task could be a prohibitive overhead.

As I mentioned above, most HPC systems have batch job systems that prefer big submissions (in relation to the average Parsl task) and that includes a preference for batch jobs that use many nodes (for example, some systems will offer a discount for batch jobs that use over a certain count - incentivising the use of a small number of many-node batch jobs, even though a pilot job workload could sometimes be scheduled more efficiently with a large number of smaller batch jobs)

In Parsl, it can be easy to get confused between the batch *jobs* which are the units of work submitted to a batch system, and correspond to blocks of workers; and *tasks* which correspond to individual app invocations. These are different things, and there is no pre-planned allocation of which task will run inside which batch job, because worker pools running inside jobs pull tasks as they are ready for more work.

3.2 Starting a block of workers

With the configuration used in *A sample task execution path* (page 11), when Parsl starts it will decide it wants some workers (how and why, see the upcoming scaling section).

The unit of allocation of workers in Parsl is called a block.

This is translated by whichever *execution provider* is being used into whatever the underlying batch system uses to represent a job or collection of worker nodes or similar. For example, with traditional HPC systems, the `SlurmProvider` will make 1 block = 1 Slurm batch job; the `KubernetesProvider` will make 1 block = 1 pod; and with the `LocalProvider`, there is no meaningful allocation of jobs and the provider will run the workers directly on the local machine with a block being the same as a Unix process.

The base class for all providers is `ExecutionProvider`, defined in parsl/providers/base.py.

As far as getting a new block of workers running, this is the most important method that a provider must implement:

```
52    @abstractmethod
53    def submit(self, command: str, tasks_per_node: int, job_name: str = "parsl.auto") -> object:
54        ...
```

The key argument here is `command`. This will be (after some mutation) be the Unix shell command that should be run on each allocated node to start the workers.

In the HighThroughputExecutor, this command is formed like this at executor start:

Todo: source code

---

and then the provider is invoked with it here:

Todo: source code

In the Task Vine executor, something similar happens at line TODO and line TODO (hrefs)

Todo: line numbers / source code link

> **Warning:** tasks_per_node is always `1` here when called by Parsl and is a fairly regular source of confusion to Parsl hackers. Maybe it should be removed. It's a vestige of an earlier time when Parsl wanted the batch system to start multiple workers on each worker node (for the long-removed IPyParallel executor). More recent executors (e.g. `HighThroughputExecutor`, `WorkQueueExecutor`, `TaskVineExecutor` and `MPIExecutor`) choose to manage (in different ways) how work is performed on a particular node rather than asking the batch system for a particular fixed number of workers.

Maybe interesting here is what is missing from the `submit` call: there is no mention of batch system queues, no mention of how many nodes to request in this block, no mention of pod image identifiers. Attributes like that are usually the same for every block submitted through (to/by?) the provider, and usually only make sense in the context of whatever the underlying batch system is: for example, a slurm job might have a queue specification and a kubernetes job might have a persistent volume specification, to be set on all jobs. These are defined in the initializer for each provider, so the provider API doesn't need to know about these specifics at all.

3.3 Launchers

Some batch systems separate allocation of worker nodes and execution of commands on worker nodes. In non-Parsl contexts that looks like: you write a batch script and submit it to slurm or PBS, and inside that batch script you prefix your application command line with something like `mpiexec` or `srun` which causes your application to run on all the worker nodes. Without that prefix, the command would run on a single node (sometimes not even in the batch allocation!)

To support this, some providers take a `launcher` parameter, which understands how to put that prefix onto the front of the relevant command. They're mostly quite simple.

All of the included launchers live in parsl.launchers.launchers and usually consist of shell scripting around something like `mpiexec` or `srun`.

3.4 Who starts processes?

Todo: a paragraph that in traditional HPC workloads, this launcher command is often responsible for starting multiple copies of your code on the same node - so if you wanted 24 cores used for an MPI code, you might use mpirun (TODO: processes_per_node param) to start 24 copies which would run in parallel. This is not how things work with parsl block workers: both the process worker pool and the WQ/TV equivalents usually manage all the tasks on a node from a single worker. So if you're feeling the temptation to make your launcher launch multiple copies of the pilot job worker, maybe there's something else going wrong? and note this is a common problem in modern times, also with OMP, where multiple layers of software think *they* are the one to spawn multiple processes/threads which leads to exponential explosion of threads. which doesn't necessarily kill your workfload but can lead to myterious performance problems. - also this section should consider *user apps* which make the same assumption (so easily

3 layers to draw diagrams about!)

Todo: above processes_per_node param

3.5 Choosing when to start or end a block

Parsl has some scaling code that starts and ends blocks as the task load presented by a workflow changes.

Todo: reference job status poller

There are three scaling strategies, which run (by default) every 5 seconds.

There are three strategy parameters defined on providers which are used by the scaling strategy: init_blocks, min_blocks and max_blocks. Broadly, at the start of a run, Parsl will launch an initial number of blocks (init_blocks) and then scale between a minimum (min_blocks) and maximum (max_blocks) number of blocks during the run.

3.5.1 The init only strategy, `none`

This strategy only makes use of the `init_blocks` configuration parameter. At the start of a workflow, it starts the specified number of blocks. After that it does not try to start any more blocks.

> **Warning:** Question: What happens if all of these initially started blocks terminate before all of the workflow's work is completed?

3.5.2 The `simple` strategy

This strategy will add more blocks when it sees that there are not enough workers.

When an executor becomes completely idle for some time, it will cancel all blocks. Even one task on the executor will inhibit cancellation - the history of this is that for abstract block-using executors, there is nothing to identify which blocks (if any) are idle. so scale out and scale in are not symmetric operations in that sense.

The scaling calculation looks at the number of tasks outstanding and compares it to the number of task slots (worker slots?) that are either running now or queued to be run.

There is a `parallelism` parameter (where?), to allow users to control the ratio of tasks to workers - by default this is 1 so Parsl will try to submit blocks to give as many worker slots as there are tasks. This does not assign tasks to particular workers: so it is common for one block to start up and a lot of the outstanding work to be processed by that block, before a second block starts which is then completely idle.

> **Warning:** Question: what does init_blocks mean in this context? Start `init_blocks` blocks then immediately scale (up or down) to the needed number of blocks?

3.5.3 The `htex_auto_scale` strategy

This is like the simple strategy for scale-out, but with better scale-in behaviour that makes use of some High Throughput Executor features: the high throughput executor knows which blocks are empty, so when there is scale-in pressure, can scale-in empty blocks while leaving non-empty blocks still running. Some prototype work has happened to try to make htex try to make blocks empty faster too, but that has not reached the production codebase.

Warning:

Todo: reference block draining problem and matthew's work.

What link here? if more stuff merged into Parsl or existing as a PR (I think there is a PR?), then the PR can be linkable. otherwise later on maybe a SuperComputing 2024 publication - but still unknown.

3.5.4 Starting workers in other ways

You can start workers without using this automated scaling: set init_blocks = min_blocks = max_blocks = 0, and then find the worker command line in the log file and run it yourself in which ever situation you want. This is good for trying things out that the provider or scaling code can't do.

The Work Queue and Task Vine executors also have their own executor specific ways for starting workers: Work Queue has a worker factory command line tool and TaskVine has a worker launch method configuration parameter.

3.6 block error handling

Todo: write error handling section (as two parts of the same feedback loop)

3.7 Worker environments

batch job environments (esp worker_init) - think about parsl requirements a bit more: Python versions, Parsl versions, installed user packages. forward reference serialization chapter.

batch job systems generally won't make the environment that your batch job providers look like the environment the submission comes from (in the case of eg. kubernetes, that's very deliberate: the job description describes the environment, not whatever ambient environment existing around the submission command. so there's a bit of tension there when you want the environment to magically look like your submission environment)

generally the python and parsl versions need to be the same as on the submit side (although people often push on this limit, and the serialization chapter will give some hints about understanding what can go wrong)

CHAPTER FOUR

ELABORATING TASKS

Earlier on, I talked about the Data Flow Kernel being given a task and mostly passing it straight on to an executor. This section will talk about the other things the Data Flow Kernel might do with a task, beyond "just run this."

In this section, I'm going to present several Parsl features which from a user-facing perspective are quite different, but they all have a common theme of the DFK doing something other than "just run this." and have some similarities in how they are implemented.

4.1 Trying tasks many times or not at all

4.1.1 Retries

When the Data Flow Kernel tries to execute a task using an Executor, this is called a try. Usually there will be one try, called try 0.

If the user has configured retries, and if try 0 fails (indicated by the executor setting an exception in the executor future then the Data Flow Kernel will retry the task. (retry without the re- is where the term "try" comes from)

Todo: if try 0 fails *OR IF THERES A SUBMIT ERROR?*

Let's have a look at the launch and retry flow in the Parsl source code. The Data Flow Kernel "launches" tasks into an executor using a method `_launch_if_ready_async`, starting at parsl/dataflow/dflow.py line 645.

(Note that the term "launch" here is distinct from the term "launch" used in the Launcher abstraction in the blocks chapter)

A task is ready to launch if it is in `pending` state and has no incomplete dependencies.

```
655 if task_record['status'] != States.pending:
656   logger.debug(f"Task {task_id} is not pending, so launch_if_ready skipping")
657   return
658
659 if self._count_deps(task_record['depends']) != 0:
660   logger.debug(f"Task {task_id} has outstanding dependencies, so launch_if_ready skipping")
661   return
```

If the code gets this far then a bit of book keeping and error handling happens, and then at line 673, the `launch_task` method will submit the task to the relevant executor and return the executor future.

```
673 exec_fu = self.launch_task(task_record)
```

… and then line 701 will attach a callback (`DataFlowKernel.handle_exec_update`) onto that executor future. This will be called when a result or exception is set on the executor future. Now `_launch_if_ready_async` can end: the Data Flow Kernel doesn't have to think about this task any more until it completes - and that end-of-task behaviour lives in `handle_exec_update`.

```
701 exec_fu.add_done_callback(partial(self.handle_exec_
    ↪update, task_record))
```

handle_exec_update is defined in dflow.py at line 323. It contains the majority of task completion code.

Task completion behaviour is defined in two cases: when the executor future contains a successful result (line 402 onwards) and when the executor future contains an exception (line 346 onwards)

The happy path of execution completing normally happens at line 408 calling DataFlowKernel._complete_task to set the AppFuture result (which is the object that lets the user see the result).

This section, though, is not about that. It is about the retry path: the exception path should be taken, and Parsl should send the task to the executor again.

In the exception case starting at line 346, the fail_cost (by default, the count of tries so far, but see the plugin section for more complications) is compared with the configured retry limit (Config.retries).

Line 368 provides the default "each try costs 1 unit" behaviour, with the 16 lines before that implementing the pluggable retry_handler.

```
368 task_record['fail_cost'] += 1
```

At line 377 and 392 there are two answers to the question: Is there enough retry budget left to do a retry?

If so, mark the task as state pending (again) at line 384 and then later on at line 454 call launch_if_ready. The task will be launched again just like before, but a bunch of task record updates have happened while processing the retry.

If there isn't enough retry budget left, then line 392 onwards marks the task as failed and marks the task's AppFuture as completed with the same exception that the executor future failed with. This is also how tasks fail In the default configuration with no retries: this code path is taken on all failures because the default retry budget is 0.

4.1.2 Checkpointing

I just talked about the Data Flow Kernel trying to execute a task many times, rather than the default of just once. Going in the other direction, there are times when Data Flow Kernel can complete a task without trying to execute it at all - namely, when checkpointing is turned on.

Note: three different names used for overlapping/related concepts: checkpointing, caching and memoization - there's no real need for using three different terms and I think as part of ongoing work here those terms could merge.

Parsl checkpointing does not try to capture and restore the state of a whole Python workflow script. Restarting a checkpointed workflow script will run the whole script from the start, but when the Data Flow Kernel receives a task that has already been run, instead of trying it even once, the result stored in the checkpoint database will be used instead.

When a workflow is started with an existing checkpointing database specified in `Config.checkpoint_files`, all of the entries in all of those files are loaded in to an in-memory `dict` stored in a `Memoizer`. This happens in `DataFlowKernel.__init__` at line 168.

When a task is ready to run, `DataFlowKernel._launch_if_ready_async` calls `DataFlowKernel.launch_task`. This will usually submit the task to the relevant executor at line 761 returning a `Future` that will eventually hold the completed result. But a few lines before at line 728, it will check the `Memoizer` to see if there is a cached result, and if so, return early with a `Future` from the `Memoizer` contained in the cached result in place of a `Future` from the executor.

```
728 memo_fu = self.memoizer.check_memo(task_record)
729 if memo_fu:
730   logger.info("Reusing cached result for task {}".
    ↪format(task_id))
```

(continues on next page)

(continued from previous page)

```
731     task_record['from_memo'] = True
732     assert isinstance(memo_fu, Future)
733     return memo_fu
```

The rest of the code still sees an executor-level future, but it happens to now come from the `Memoizer` rather than from the relevant `Executor`.

If a task is actually run by an executor (because it was not available in the existing checkpoint database), then on completion (in `DataFlowKernel.handle_app_update` which is another callback, this time run when an AppFuture is completed) `DataFlowKernel.checkpoint` will be invoked to store the new result into the `Memoizer` and (depending on configuration) the checkpoint database, at line 566 onwards.

> **Warning:** `handle_app_update` is a bit of a concurrency wart: because it runs in a callback associated with the AppFuture presented to a user, the code there won't necessarily run in any particular order wrt user code and so it can present some race conditions. This code could move into end-of-task completion handling elsewhere in the DFK, perhaps. See issue #1279.

Todo: do I want to talk about how parameters are keyed here? YES Note on ignore_for_cache and on plugins (forward ref. plugins)

Todo: make a forward reference to *Serializing tasks and results with Pickle* (page 55) section about storing the result (but not the args)

Todo: task identity and dependencies: there is a notion of "identity" of a task across runs here, that is different from the inside-a-run identity (aka the task id integer allocated sequentially) – it's the hash of all arguments to the app. So what might look like two different invocations fut1 = a(1); fut2 = a(1) to most of Parsl, is actually two invocations of "the same" task as far as checkpointing is concerned (because the two invocations of `a` have the same argument). Another subtlety here is that this identity can't be computed (and so we can't do any checkpoint-replacement) until the dependencies of a task have been completed - we have to run the dependencies of a task T (perhaps themselves by checkpoint restore) before we can ask if task T itself has been checkpointed.

4.2 Modifying the arguments to a task

In the previous section I talked about choosing how many times to execute a task, and maybe replacing the whole executor layer execution with something else. In this section, I'll talk about modifying the task before executing it, driven by certain special kinds of arguments.

4.2.1 Dependencies

Parsl task dependency is mediated by futures: if a task is invoked with some `Future` arguments, that task will eventually run when all of those futures have results, with the individual future results substituted in place of the respective `Future` arguments. (so you can use *any* `Future` as an argument - it doesn't have to be a Parsl `AppFuture`)

Earlier on (in the retry section) I talked about how `DataFlowKernel._launch_if_ready_async` would return rather than launch a task if `DataFlowKernel._count_deps` counted any outstanding futures.

This happens in a few stages:

- as part of `DataFlowKernel.submit` (the entry point for all task submissions), at line 1078, `DataFlowKernel._gather_all_deps` examines al of the arguments for the task to find `Future` objects that the task depends on. These are then stored into the task record.

```
1078 depends = self._gather_all_deps(app_args, app_
     ↪kwargs)
1079 logger.debug("Gathered dependencies")
1080 task_record['depends'] = depends
```

- In order to get launch if ready to be called when all the futures are done, each future has a callback added which will invoke launch if ready
- inside `_launch_if_ready_async`, `DataFlowKernel._count_deps` loops over the Future objects in `task_record['depends']` and counts how many are not done. If there are any not-done futures, `_launch_if_ready_async` returns without launching:

```
655 if task_record['status'] != States.pending:
656   logger.debug(f"Task {task_id} is not pending,
    ↪ so launch_if_ready skipping")
657   return
658
659 if self._count_deps(task_record['depends']) !=␣
    ↪0:
660   logger.debug(f"Task {task_id} has␣
    ↪outstanding dependencies, so launch_if_ready␣
    ↪skipping")
661   return
662
663 # We can now launch the task or handle any␣
    ↪dependency failures
664
665 new_args, kwargs, exceptions_tids = self._
    ↪unwrap_futures(task_record['args'],
```

(continues on next page)

(continued from previous page)

```
666                         ␣
↪            task_record['kwargs'])
```

So `_launch_if_ready_async` might run several times, once for every dependency `Future` that completes. When the final outstanding future completes, that final invocation of `_launch_if_ready_async` will see no outstanding dependencies - the task will be ready in the "launch if ready" sense.

At that point, the DFK unwraps the values and/or errors in all of the dependency futures. `_unwrap_futures` takes the full set of arguments (as a sequence of positional arguments and a dictionary of keyword arguments) and replaces each `Future` with the value of that `Future`. The arguments for the task are replaced with these unwrapped arguments.

It is possible that a `Future` contains an exception rather than a result, and these exceptions are returned as the third value, `exceptions_tids`. If there are any exceptions here, that means one or more of the dependencies failed and we won't be able to execute this task. So the code marks that code as failed (in a `dep_fail` state to distinguish it from other failures).

Otherwise, task execution proceeds with this freshly modified task.

> **Warning:** how can we meaningfully return new_args and kwargs if there were any exceptions?

4.2.2 File staging

Another modification to the arguments of a task happens with the file staging mechanism. In the dependency handling code, special meaning is attached to `Future` objects. In the file staging code, special meaning is attached to `File` objects.

The special meaning is that when a user supplies a `File` object as a parameter, then Parsl should arrange for file staging to happen before the task runs or after the task completes.

> **Warning:** The terminology around file staging is a bit jumbled. There is a historical conflation of "files" and "data" so file staging is sometimes called data staging, and a big piece of staging code is called the "data manager", despite being focused on files not other data such as Python objects. In configuration, file staging providers are configured using a "storage access" parameter.

In `DataFlowKernel.submit`, at task submit time, the arguments are examined for file objects, and the file staging code can make substitutions. Like dependencies, substitutions can happen to positional and keywords arguments, but the function to be executed can be substituted too!

```
1058 # Transform remote input files to data futures
1059 app_args, app_kwargs, func = self._add_input_
     ↪deps(executor, app_args, app_kwargs, func)
1060
1061 func = self._add_output_deps(executor, app_args,␣
     ↪app_kwargs, app_fu, func)
1062
1063 logger.debug("Added output dependencies")
1064
1065 # Replace the function invocation in the TaskRecord␣
     ↪with whatever file-staging
1066 # substitutions have been made.
```

(continues on next page)

(continued from previous page)

```
1067 task_record.update({
1068             'args': app_args,
1069             'func': func,
1070             'kwargs': app_kwargs})
```

This supports two styles of file staging:

A file staging provider (invoked inside `_add_input_deps` or `_add_output_deps`) can submit staging tasks to the workflow. For staging in, it can create stage-in tasks and substitute a `Future` for the original `File` object. These futures will then be depended on by the dependency handling code which runs soon after. For outputs, tasks can be submitted which depend on the task completing, by depending on `app_fu`. With this style of staging, file transfers are treated as their own workflow tasks and so, for example, you can see them as separate tasks in the monitoring database.

The other style of file staging runs as a wrapper around the application function. A file staging provider replaces the function defined by the app with a new function which performs any stage in, runs the original app function, performs any stage out and returns the result from the app function. This style is aimed at situations where staging must happen close to the task - for example, if there is no shared filesystem between workers, then it doesn't make sense to stage in a file on one arbitary worker and then try to use it on another arbitrary worker.

Parsl has example HTTP staging providers for both styles so you can compare how they operate. These are in parsl/data_provider/http.py.

Todo: maybe a simple DAG to modify here based on previous staging talks

Warning:

Todo: note about app future completing as soon as the value is available and not waiting till stage-out has happened - See issue #1279.

4.2.3 Rich dependency resolving

Todo: including rich dependency resolving - but that should be an onwards mention of plugin points? and a note about this being a common mistake. but complicated to implement because it needs to traverse arbitrary structures. which might give a bit of a tie-in to how `id_for_memo` works)

Note: Future development: these can look something like "build a sub-workflow that will replace this argument with the result of a sub-workflow" but not quite: file staging for example, has different modes for outputs, and sometimes replaces the task body with a new task body, rather than using a sub-workflow. Perhaps a more general "rewrite a task with different arguments, different dependencies, different body" model?

4.3 Wrapping tasks with more Python

The file staging section talked about replacing the user's original app function with a wrapper that does staging as well as executing the wrapped original function.

That's a common pattern in Parsl, and happens in at least these places:

- Bash apps, which execute a unix command line, are mostly implemented by wrapping `remote_side_bash_executor` (in

parsl/app/bash.py) around the user's Python app code. On the remote worker, that wrapper executes the user's Python app code to generate the command line to run, and then executes that as a unix command, turning the resulting unix exit code into an exception if necessary.

That means no part of Parsl apart from the `bash_app` decorator and corresponding `BashApp` have any idea what a bash app is. The rest of Parsl just sees Python code like any other task.

- When resource monitoring is turned on, the DFK wraps the users task in a monitoring wrapper at launch, at parsl/dataflow/dflow.py line 74. This wrapper starts a separate unix process that runs alongside the worker, sending information about resource usage (such as memory and CPU times) back to the monitoring system.
- The python_app timeout parameter is implemented as wrapper which starts a thread to injects an exception into an executing Python app when the timeout is reached. See parsl/app/python.py line 18.
- All apps are wrapped with `wrap_error`. This wrapper (defined in parsl/app/errors.py line 134) catches exceptions raised by the user's app code and turns it into a RemoteExceptionWrapper object. This is intended to make execution more robust when used with executors which do not properly handle exceptions in running tasks. The RemoteExceptionWrapper is unwrapped back into a Python exception as part of the Data Flow Kernel's result handling.

Note: This is one of the hardest (for me) conceptual problems with dealing generally with MPI. What does an MPI "run this command line on n ranks" task interface look like when we also want to say "run this arbitrary wrapped Python around a task"?

4.4 Join apps: dependencies at the end of a task

Join apps are a way to launch new tasks (and other Future-like) behaviour inside a workflow, avoiding blocking use of `Future.result()` which can hurt concurrency.

The original idea for them was to allow "sub-workflows" to be launched as results became available, when the sub-workflow couldn't even be described until some result is available - for example, we need to launch n tasks but we don't know what n is until later.

Later on, it turned out they can be used to calls into other execution systems that return `Future` objects. For example, here's a blog post about submitting into Globus Compute using join apps.

Users make use of this by writing some Python code inside a join app that launches tasks and returns the Futures of those tasks. When this code finishes, the task enters a new state (not used for other apps) called `joining` which looks a bit like dependency handling, but at the result end of the task. Parsl will wait until all of the returned futures have completed and then return the contents of those futures as the result of the task.

The `join_app` decorated is implemented as a variant of the `python_app` decorator that sets an additional bit to indicate it is a join app and forces execution to happen on the `_parsl_internal` thread pool executor.

The user's app code is forced to execute onto the `parsl_internal` because it must run in the same process as the Data Flow Kernel: it wants wants submit tasks to the same Data Flow kernel or something else running in the main workflow process (rather than a limited worker environment) and `Future` objects don't make sense to move across the network between processes: they're a dynamic reflection of some local execution state.

That join flag finds its way into the TaskRecord. It doesn't affect execution of the app until the code path in `handle_exec_update` which

deals with successful task completion (at parsl/dataflow/dflow.py line 134).

```
134 if not task_record['join']:
135   self._complete_task(task_record, States.exec_done,
    ↪ res)
136   self._send_task_log_info(task_record)
137 else:
138   # This is a join task, and the original task's␣
    ↪function code has
139   # completed. That means that the future returned␣
    ↪by that code
140   # will be available inside the executor future,␣
    ↪so we can now
141   # record the inner app ID in monitoring, and add␣
    ↪a completion
142   # listener to that inner future.
143
144   joinable = future.result()
145
146   # Fail with a TypeError if the joinapp python␣
    ↪body returned
147   # something we can't join on.
148   if isinstance(joinable, Future):
149     self.update_task_state(task_record, States.
    ↪joining)
150     task_record['joins'] = joinable
151     task_record['join_lock'] = threading.Lock()
152     self._send_task_log_info(task_record)
153     joinable.add_done_callback(partial(self.handle_
    ↪join_update, task_record))
```

In the normal (non-join-app) case, that code will complete the task (for example by setting the `AppFuture` result. In the join case, the task instead goes into a new `joining` state, and further completion will happen in another callback, when the joinable Future is completed. There is another case right after to handle the app returning a list of Futures.

That `handle_join_update` callback looks quite like dependency handling of `launch_if_ready`: it will run once for each joinable Future, it checks if all the joinable Futures are completed, and moves the task onto the next state if so - in this case, marking the task as complete (vs the dependency behaviour of launching the task)

Todo: earlier on there should be a state graph. then here the same graph with the joining state.

See also:

If you're interested in functional programming, join apps basically treat futures as a forming a monad. The term "join" comes from monadic join that takes `Future[Future[X]] -> Future[X]` which is the extra behaviour that join apps add onto the end of regular Python apps. If none of that makes sense, don't worry: you don't need category theory to use `join_app`!

4.4.1 Putting these all together

Todo: Summarise by me pointing out that in my mind (not necessarily in the architecture of Parsl) that from a core perspective these are all quite similar, even though the user effects are all very different. Which is a nice way to have an abstraction. And maybe that's an interesting forwards architecture for Parsl one day…

CHAPTER FIVE

UNDERSTANDING THE MONITORING DATABASE

Parsl can store information about workflow execution into an SQLite database. Then you can look at the information, in a few different ways.

5.1 Turning on monitoring

Here's the workflow used in *A sample task execution path* (page 11), but with monitoring turned on:

```
import parsl

def fresh_config():
  return parsl.Config(
    executors=[parsl.HighThroughputExecutor()],
    monitoring=parsl.MonitoringHub(hub_address =
↪"localhost")
  )

@parsl.python_app
def add(x: int, y: int) -> int:
  return x+y
```

(continues on next page)

(continued from previous page)

```
@parsl.python_app
def twice(x: int) -> int:
  return 2*x

with parsl.load(fresh_config()):
  print(twice(add(5,3)).result())
```

Compared to the earlier version, the changes are adding `monitoring=` parameter to the Parsl configuration, and adding an additional app `twice` to make the workflow a bit more interesting.

After running this, you should see a new file, `runinfo/monitoring.db`:

```
$ ls runinfo/
000
monitoring.db
```

This new file is an SQLite database shared between all workflow runs that use the same `runinfo/` directory.

5.2 Using monitoring information

There are two main approaches to looking at the monitoring database: the prototype `parsl-visualize` tool, and Python data analysis.

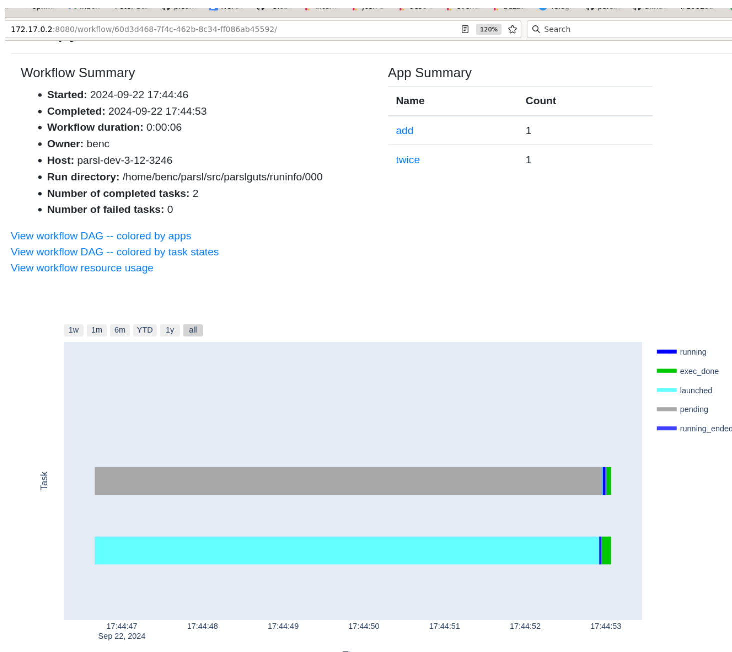
5.2.1 parsl-visualize web UI

Parsl comes with a prototype browser-based visualizer for the monitoring database.

Start it like this, and then point your browser at the given URL.

```
$ parsl-visualize
 * Serving Flask app 'parsl.monitoring.visualization.app'
 * Debug mode: off
WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.
 * Running on http://127.0.0.1:8080
Press CTRL+C to quit
```

Here's a screenshot, showing the above two-task workflow spending most of its 5 second run with the `add` task in `launched` state (waiting for a worker to be ready to run it), and the `twice` task in `pending` state (waiting for the `add` task to complete).



I'm not going to go further into `parsl-visualize` but you can run

your own workflows and click around to explore.

5.2.2 Using data frames

A different approach preferred by many data-literate users is to treat monitoring data like any other Python data, using Pandas.

This example loads the entire task table (for all known workflows) into a data frame and then extracts the task completion times using Pandas notation:

```
import pandas as pd
import sqlite3

c = sqlite3.connect("runinfo/monitoring.db")
df = pd.read_sql_query("SELECT * FROM task", c)
c.close()

print(df['task_time_returned'])
```

```
$ python3 panda_mon.py
0    2024-09-22 17:44:52.947501
1    2024-09-22 17:44:53.005619
Name: task_time_returned, dtype: object
```

Todo: one example of plotting

5.3 What is stored in the database?

Todo: deeper dive into workflow/tasks/try table schema - not trying to be comprehensive of all schemas here but those three are a good set to deal with

The monitoring database SQL schema is defined using SQLAlchemy's object-relational model at parsl/monitoring/db_manager.py line 132 onwards.

> **Warning:** The schema is defined a second time in parsl/monitoring/visualization/models.py line 12 onwards. See issue #2266 for more discussion.

These tables are defined:

Todo: the core task-related tables can get a hierarchical diagram workflow/task/try+state/resource

---

- `workflow` - each workflow run gets a row in this table. A workflow run is one call to `parsl.load()` with monitoring enabled, and everything that happens inside that initialized Parsl instance.
- `task` - each task (so each invocation of a decorated app) gets a row in this table
- `try` - if/when Parsl tries to execute a task, the try will get a row in this table. As mentioned in *Elaborating tasks* (page 33), there might not be any tries, or there might be many tries.
- `status` - this records the changes of task status, which include changes known on the submit side (in `TaskRecord`) and changes which are not otherwise known to the submit side: when a task starts and ends running on a worker. You'll see `running` and `running_ended` states in this table which will never appear in the `TaskRecord`. One `task` row may have many `status` rows.
- `resource` - if Parsl resource monitoring is turned on (TODO: how?), a sub-mode of Parsl monitoring in general, then a resource monitor process will be placed alongside the task (see *Elaborating tasks* (page 33)) which will report things like CPU

time and memory usage periodically. Those reports will be stored in the resource table. So a try of a task may have many resource table rows.

- `block` - when the scaling code starts or ends a block, or asks for status of a block, it stores any changes into this table. If enough monitoring is turned on, the block where a try runs will be stored in the relevant `try` table row.
- `node` - this one is populated with information about connected worker pools with htex (and not at all with other executors), populated by the interchange when a pool registers or when it changes status (disconnects, is set to holding, etc)

CHAPTER

SIX

SERIALIZING TASKS AND RESULTS WITH PICKLE

Todo: some visualizations for pieces of this could be loosely disassembled pickle bytecode - otherwise lacking in code-level visualization

In a lot of the code examples so far, Python objects go from one piece of code ot another as regular arguments or return values. But in a few places, those objects need to move between Python processes and usually that is done by turning them into a byte stream at one end using Python's built in `pickle` library, sending that byte stream, and turning the byte stream back into a new Python object at the other end.

Some of the places this happens:

- sending task definitions (functions and arguments) from the High Throughput executor in the users workflow script to the process worker pool; and sending results back the other way.
- Storing results in the checkpoint database, to be loaded by a later Python process, and also in computing object equality for looking up checkpoint results - see *Elaborating tasks* (page 33).
- Sending monitoring messages

- Communication between some different Python processes - both high throughput executor and the monitoring system involve multiple processes, and they often send each other objects (often dictionaries) over network and interprocess communication. Sometimes without it being explicit (for example, Python's `multiprocessing` library makes heavy use of `pickle`). ZMQ's send_pyobj / recv_pyobj uses `pickle` to turn the relevant Python object into a bytestream that can be sent over ZMQ, and back.

A lot of the time, this works pretty transparently and doesn't need much thought: for example, a Python integer object `123456` is easy to pickle into something that comes out the other end as an equivalent object.

But, there are several situations in Parsl where there are complications, and it can help to have some understanding of what is happening inside `pickle` when trying to debug things - rather than trying to regard `pickle` as a closed magical library.

intro should refer to not regarding this as magic, despite most people desperately hoping it is magic and then not trying to understand whats happening. this is needs a bit of programming language thinking, way more than routing "tasks as quasi-commandlines"

I'll use the term pickling and serializing fairly interchangeably: serialization is the general word for turning something like an object (or graph of objects) into a stream of bytes. Pickling is a more specific form, using Python's built in Pickle library.

As I mentioned in *A sample task execution path* (page 11), when the High Throughput Executor wants to send a function invocation to a worker, it serializes the function and its arguments into a byte sequence, and routes that to a worker, where that byte sequence is turned back into objects that are in some sense equivalent to the original objects. Task results follow a similar path, in reverse.

That serialization is actually mostly pluggable, but basically everyone uses some variant of pickle (most often the dill library) because that's the default and there isn't much reason to change.

For most things that look like simple data structures, pickling is pretty simple. For example, almost anything that you can imagine some obvious representation in JSON, plain pickle won't have a problem.

There are a few areas where it helps to have some deeper understanding of whats going on, so that you don't run into "mystery pickling errors because the magic is broken."

6.1 Tiny pickle tutorial

This is a simple example of Pickle that is enough for most use cases: one function `dumps` turns a fairly arbitrary Python object into a byte sequence, and another `loads` turns it back into an object again. The point of doing this is that moving a byte sequence around is a much clearer, more flexible operation than sending an arbitrary object around.

```
b: bytes = pickle.dumps(some_obj)

# send b somewhere through time and space

some_object.loads(b)
```

6.2 Functions

6.2.1 Using pickle

You have probably got some notion of what it means to send a function across the network, and those preconceptions are almost definitely not how Parsl does it. So you need to put those preconceptions aside.

`pickle` on its own cannot send the definition of functions. If you try to pickle a function named `mymodule.f`, the resulting pickle contains the equivalent of `from mymodule import f`.

So in order for this to unpickle in the Python process at the other end, that statement `from mymodule import f` needs to work. The usual Python reasons why that statement might not work apply to unpickling. For example, `mymodule` needs to be installed, and needs to be enough of a compatible version to import `f`.

Todo: the "function is in __main__ which is different remotely"

Todo: f does not have a name

This can happen in a few ways: the biggest one for Parsl is that a python-app decorated function (yes, that's every app defined using a decorator) - the function body won't be the same as the value assigned to the app name variable. because that vairable is used for the PythonApp object, not the underlying function.

That can be worked around by letting a function get a global name, using a variant of the decorator syntax I talked about n the first chapter:

```
def myfunc(a,b):
  return a+b

myapp = python_app(myfunc)
```

now the underlying function is available with `from wherever import myfunc` and the Parsl app equivalent can be invoked with `myapp(3,4)`.

Another situation where a function does not have a global name is when it is defined as a closure inside another function:

```
def add_const(n):
  def myfunc(a,n):
    return a+n
```

(continues on next page)

(continued from previous page)

```
myapp = python_app(add_const(7))
```

This is pretty common in certain functional styles of Python programming. One way to think about how it is a problem is to try to write an `import` statement to import the underlying function for `myapp`.

6.2.2 Using dill

Parsl makes extensive use of the dill library. Dill aims to let you serialize all the bits of Python that pickle cannot deal with, building on top of the Pickle protocol.

For functions, it tries to address the above problems by using its own function serialization, in circumstances where it has decided that the default pickle behaviour will not work (sometimes deciding correctly, sometimes using a heuristic which can go wrong).

`dill` function serialization does not use the `pickle` method of sending by reference. Instead it sends the Python bytecode for the function. This does not need the function to be importable at the receiving end. Some downsides of this approach are that Python bytecode is not compatible across Python releases, and `dill` does not contain any protection for this: executing bytecode from a different Python version can result in the executing Python process exiting or worse, perhaps even incorrect results. Functions serialized this way can also sometimes bring along a lot of their environment (if dill decides that environment will also not be available remotely) which can result in extremely large serialized forms, and occasionally crashes due to serializing the unserializable - see Parsl issue #2668 for example.

Todo: URL for Python bytecode/virtual machine documentation?

Todo: backref/crossref the worker environment section - it could point here as justification/understanding of which packages should be installed.

6.2.3 Dill vs Pickle

dill and pickle will between them usually be able to serialize a function one way or the other, but it can be quite subtle which method was chosen, and the two methods have very different characteristics:

- pickle: if we can import the function from an installed library. works across python versions
- dill: if we cannot import the function from an installed library. likely to cause random behaviour across python versions.

subtleties of chosing between the two include where a file is imported from (so that dill might decide it is an installed library, which can be serialized as an `import`, or might decide it is not an installed library but instead user code that it does not expect to be available remotely and so must be sent as bytecode)

Todo: also mention cloudpickle as a dill-like pickle extension. They are both installable alongside each other... and people mostly haven't given me decent argumetns for cloudpickle because people don't dig much into understanding whats going on.

6.3 Exceptions

The big deal here is with trying to use package specific classes, only having them installed on the remote side, but then not realising that an exception being raised is also a package specific class.

Environments have to be consistent all over. That doesn't mean they have to be identical. But problems arise when people try to use insufficiently consistent environments: things work OK most of the time because no "worker side only" objects are sent around,

Custom classes are also usually sent by reference, in the same way that Python sends functions.

Todo: i think there's a funcx approach to this that i could link to that turns exceptions into strings, which are basic pickle data types we should always be able to unpickle. see issue #3474. You lose the ability to catch specific exceptions (at least in the standard Python way).

6.4 Some objects don't make sense to send to other places

Objects that are "data like" make sense to pickle. An intuitive way to think about "data like" is "could you write down the value of the object on a piece of paper?".

Some objects don't represent that - for example a Thread object represents a running thread in a particular Python process. Ask yourself what it means to pickle/unpickle that object into a different Python process, perhaps on a different machine? `Future` is another example of that, and maybe the most common to encounter when getting your head around launching tasks inside other tasks (see join apps)

In between there are more interesting objects that try to do interesting things with the serialization process .. ProxyStore is probably the most interesting example of that.

See also:

I've talked about Pickle in more depth and outside of the Parsl context at PyCon Lithuania: The Ghosts of Distant Objects

Serialising functions is a hard part of programming languages, especially in a language that wasn't designed for this, and parsl is constantly pushing up against those limits. have a look at https://www.unison-lang.org/ if you're interested in languages which are trying to do this from the start.

CHAPTER

SEVEN

MODULARITY AND PLUGINS

7.1 Motivation

7.1.1 why

Parsl exists as a library within the python ecosystem. Python *exists* as a user-facing language, not an internal implementation language. Our users are generally Python users (of varying degree of experience) and we can make use of that fact.

structuring of code within the parsl github repo. "why" includes sustainability work on different quality of code/maintenance. different quality includes things like "this piece of code is well tested, or tested by this environment". different levels of support for different contributions.

it's also a place to plug in "policies" - that is user-specified decisions (such as how to retry, using retry handlers) that take into account the ability of our users to write Python code as policy specifications.

place for supporting other non-core uses: for example Globus Compute makes use of the plugin API to use only htex and the lrm provider parts of Parsl, and can do that because of the plugin API, where it becomes its own plugin host for the relevant plugins.

place for research/hacking - eg. want to do some research on doing X

with workflows. Parsl has a role there as being the workflow system that exists that you can then modify to try out X, rather than writing your own toy workflow system. want to try out an idea. (example for parslfest: matthew chungs work involved very minimal changes to Parsl - including a new plugin interface! - for a nice outcome). two things there: beneficial for the code to be modular (even within the same repo) so that you only need to understand the pieces you want to hack on, with less understanding needed of less relevant parts. ability to share add-ons without people having to patch parsl (although in reality that doesn't really happen)

7.1.2 how

if there's a decision point that looks like a multi-way if statement - having a bunch of choices is a suggestion that choices you might not have implemented might also exist, and someone might want to put those in. various plugin points then look like "expandable if" statements. a good contrast is the launcher plugin interface, vs the hard-coded MPI plugin interface (cross reference issue to fix that), described in the context of pluggability and needing to modify parsl source code.

use the phrase "dependency injection"

7.1.3 rest

this is an architectural style rather than an API

there have been a few places in earlier sections where i have talked (in different ways) about plugging in different pieces of code - the biggest examples being providers and executors.

The big examples that lots of people encounter for this section are providers, because this is a big part of describing the unique environment of each different system; and executors, because one of the ways that other research groups like to collaborate with big code chunks is by Contributing interfaces so Parsl's DFK layer can submit to their

own execution system rather than using the High Throughput Executor. The biggest example of that is Work Queue, but there are several other executors in the codebase.

Doing that sort of stuff is what I'd expect as part of moving from being a tutorial-level user to a power user.

7.2 An example: providers

[modularity example] In the blocks section, (TODO crossref) I showed how different environments need different providers and launchers, but that the scaling code doesn't care about how those providers and launchers do their work. This interface is a straightforward way to add support for new batch systems, either in the Parsl codebase itself, or following the interface but defined outside of the Parsl codebase.

who cares about what

the API

7.3 An example: retry policies

Python exceptions - a user knows more about the exceptions than infrastructure does. That's why Python lets you catch certain exceptions and deal with them in different ways.

Parsl propagates those exceptions to the user via the relevant `AppFuture`, but by that time it's too late to influence retries.

a simple policy: if i get a worker or manager failure, retry 3 times, because this might be transient. if i get a computation failure (let's say divide by zero) then do not retry because i expect this is "permanent". this is something that doesn't belong in the Parsl codebase: it is application specific behaviour. So we're using plugin concept here to allow users to attach their application code into parsl in a way that cannot be done through the main task interface.

7.4 All the plugin points I can think of

Todo: for each, a sentence or two, and a source code reference

- executors - you've got a function and arguments and want to run the function with the arguments. but probably somewhere else, queued or managed in some way. That's what an executor does, by providing the DataFlowKernel with a submit call:

  https://github.com/Parsl/parsl/blob/3f2bf1865eea16cc44d6b7f8938a1ae1781c61fd/parsl/executors/base.py#L80

```
80  @abstractmethod
81    def submit(self, func: Callable, resource_
↪specification: Dict[str, Any], *args: Any,␣
↪**kwargs: Any) -> Future:
```

  The big example here is using Work Queue to get access to work queue's resource allocation language which is much more expressive than the high throughput executor's worker slot mechanism. There are other executors here too though, built on radical pilot, flux, and task vine.

- providers - addressed in previous section
- launchers
- (scheduled for removal) Channels - so I won't describe them
- retry handlers - this is a place to encapsulate user knowledge about if a task should be retried, and if so how much. By default the cost of a task retry is 1 unit.

  https://github.com/Parsl/parsl/blob/3f2bf1865eea16cc44d6b7f8938a1ae1781c61fd/parsl/config.py#L113

retry_handler: Optional[Callable[[Exception, TaskRecord], float]]

A retry handler is a function like this:

```
def my_retry_handler(e: Exception, t:␣
↪TaskRecord) -> float:
```

which is called by the Data Flow Kernel when a task execution fails. It can look at both the exception from that failing task execution, and at `TaskRecord` (including the function and arguments) and decide in some application specific way how much this should cost.

The standard example here is distinguishing between exceptions that might be worth retrying (such as a crashed worker) and exceptions that are less likely to succeed if run a second time (for example, some application reported calculation error)

- memoizer key calculator (id_for_memo)

  When checkpointing to disk (as mentioned in *Elaborating tasks* (page 33)), Parsl stores a record for each task that has been completed. Each task is identified by a hash of the task arguments (and some other stuff). On a re-run, the task is hashed again and that hash is looked up in the checkpoint database. It isn't possible to compute a meaningful equality-like hash for arbitrary Python objects. Parsl uses a single dispatch function `id_for_memo` to compute meaningful equality hashes for several built-in Python types, and this is the way to plug in hash computation for other types.

  Here's an example from parsl/dataflow/memorization at line 61 which recursively defines how to hash a list. `id_for_memo.register` can be called a user workflow script to register more types.

```
61 @id_for_memo.register(list)
62 def id_for_memo_list(denormalized_list: list,␣
   ↪output_ref: bool = False) -> bytes:
```

(continues on next page)

(continued from previous page)

```
63    if type(denormalized_list) is not list:
64        raise ValueError("id_for_memo_list␣
↪cannot work on subclasses of list")
65
66    normalized_list = []
67
68    for e in denormalized_list:
69      normalized_list.append(id_for_memo(e,␣
↪output_ref=output_ref))
70
71    return pickle.dumps(normalized_list)
```

- file staging

  I talked about file staging in *Elaborating tasks* (page 33), with staging providers allowed to launch new tasks and replace the body function of a task. The `Staging` interface in parsl/data_provider/staging.py provides methods to do that.

- default stdout/stderr name generation

- Rich dependency handling

  Sometimes it is nice to pass arguments that are structures which contain futures, rather than the argument directly being Futures - for example, a list or dictionary of futures. Parsl's default dependency handling won't see those futures hidden inside other structures, and so will neither wait for them to be ready, not substitute in their values.

  Parsl's dependency resolver hook lets you add in richer dependency handling by substituting in your own code to find and replace Futures inside task arguments. As an example, the `DEEP_DEPENDENCY_RESOLVER` defined in parsl/dataflow/dependency_resolvers.py line 111 provides an implementation which can be extended by type (like `id_for_memo` above).

Todo: ref back to *Elaborating tasks* (page 33) if I write that section

---

- serialization - although as hinted at in *Serializing tasks and results with Pickle* (page 55), Pickle is also extensible and that is usually the place to plug in hooks.

Todo: link to serialization interface, and to pickle documentation for pickle extensibility

---

- High Throughput Executor interchange manager selectors - https://github.com/Parsl/parsl/blob/3f2bf1865eea16cc44d6b7f8938a1ae1781c61fd/parsl/executors/high_throughput/manager_selector.py - this is the beginning of a plugin interface to choose how tasks and worker pools are matched together in the interchange.

CHAPTER

EIGHT

COLOPHON

Marked-up in reStructuredText

Rendered with sphinx

Edited with `vi` and vscode

Sitting in Berlin

This text was prepared against Parsl version 2024.09.02

INDEX

Symbols

_parsl_internal executor, 44

A

AppFuture, 15
apps
- bash, 43
- join, 44
- python, 12

B

bash_app, 43
batch jobs, 21
blocks, 21, 24

C

checkpointing
- id_for_memo, 65
- serialization, 55

D

Data Flow Kernel, 15
data frame, 52
decorators, 12
DFK, 15
dill, 59

F

File, 40
Futures
- serialization, 44, 61

G

Globus Compute, 16, 44, 60
God object, 15

H

High Throughput Executor
- htex_auto_scale, 29
- interchange, 18
- process worker pool, 19

htex_auto_scale, 29

I

id_for_memo, 65
interchange, 18

J

join_app, 44

K

kubernetes, 24

L

launch, 34
launchers, 26
library
    pandas, 52
    sqlite3, 47

M

monads, 44
monitoring, 47
    configuration, 49
    pandas, 52
    parsl-visualize, 50
    resource wrapper, 43
    schema, 52
MonitoringHub, 49
mpiexec, 27
mpirun, 27

P

pandas, 52
parsl-visualize, 50
pilot jobs, 19, 23
plugins
    checkpointing, 35
    file staging
        providers, 40
    launchers, 26
    providers, 24
    retry_handler, 33
providers, 24, 65
Python apps, 12
python_app, 12

R

retries, 33
    policy, 65

S

scaling
    strategy, 28
serialization
    checkpointing, 55
    dill, 59
    functions, 57
    Futures, 44, 61
slurm, 24
SQL, 47
SQLite, 47
srun, 27
strategy
    scaling, 28
sub-workflows, 44

T

task, 15
TaskRecord, 15
    depends, 33, 38
    fail_cost, 33
    join, 44
    joins, 44
timeout, 43
tries, 33

U

user workflow process, 9

W

worker pool, 19

Z

ZeroMQ, 16
ZMQ, 16