Parsl Guts

Release 1.3.0-dev

Ben Clifford

Aug 21, 2024

CONTENTS

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 is a contrast to the user guide, which focuses on what Parsl looks like from the outside.

This text is not intended to be a comprehensive guide to all parts of the Parsl codebase: there is plenty more to learn about.

I'll try to include links to relevant external resources: source code, other Parsl documentation, community talks, github issues, papers and other research work.

CHAPTER

TWO

A SAMPLE TASK EXECUTION PATH

the codepath of a task from invoking an app to running on an HTEX worker, and back again

this assumes a basic hpc-like environment. so lets have a sample configuration involving htex and the slurm provider and not much else.

TODO: sample configuration, as defaulting as possible:

here's an app that adds two numbers

TODO: app definition

and now we can initialise a Parsl context manager, invoke the app and wait for its result.

TODO: with/invoke code blocks

Deliberately nothing fancy there: this is getting-started levels of Parsl use.

Now lets pick apart what happens. I'm going to ignore: the startup/shutdown process (what happens with parsl.load() and what happens at the end of the with block), and I'm going to defer batch system interactions to another section (TODO: hyperlink blocks)

2.1 a decorated app

first lets look at what we defined when we defined our app. Normally `Function definitions` defines a function (or a method) in Python. With the python_app decorator, instead that defines a PythonApp object. That's something we can invoke, like a function, but it's going to do something parsl specific.

look at definition of python app and see the `__call__` definition. when you invoke an app myapp(5,6), then the relevant `PythonApp.__call__` method is invoked. the decorator stashed away the actual user defined body of code in an attribute, and it can pass that into `__call`

2.2 the data flow kernel

we can have a look at that method and see that to "invoke an app", we call a method on the DataFlowKernel (DFK), the core object for a workflow (historically following the God-object antipattern).

inside the DFK:

- create a task record and an AppFuture, and return that AppFuture to the user

Then asynchronously:

- perform "elaborations" - see elaborations chapter
- send the task to an Executor (TODO:hyperlink class docstring). in this case we aren't specifying multiple executors, so the task will go to the default single executor which is an instance of the High Throughput Executor (TODO: hyperlink class docstring) - which generates an executor level future
- wait for completion of execution (success or failure) signlled via the executor level future
- a bit more post-execution elaboration

- set the AppFuture result

so now lets dig into the high throughput executor. the dataflow kernel hands over control to whichever executor the user configured (the other options are commonly the thread pool executor (link) and work queue (link) although there are a few others included). but for this example we're going to concentrate on the high throughput executor. If you're a globus compute fan, this is the layer at which the globus compute endpoint attaches to the guts of parsl - so everything before this isn't relevant for globus compute, but this bit about the high throughput executor is.

The data flow kernel will have performed some initialization on the high throughput executor when it started up, in addition to the user-specified configuration at construction time - (TODO: perhaps this is in enough of one place to link to in the DFK code?). for now, I'm going to assume that all the parts of the high throughput executor have started up correctly.

htex consists of a small part that runs in the user workflow process (TODO: do I need to define that as a process name earlier on in this chapter? it's somethat that should be defined and perhaps there should be a glossary or index for this document for terms like that?) and several other processes.

The first process in the interchange (TODO: link to source code). This runs on the same host as the user workflow process and offloads task and result routing.

Beyond that, on each worker node on our HPC system, a copy of the process worker pool will be running. These worker pools connect back to the interchange using two network connections (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)

so inside htex.submit: we're going to:

- serialize the details of the function invocation into a sequence of bytes. this is non-trivial even though everyone likes to believe

it is magic and simple. In a later chapter I'll talk about this in much more depth (TODO: link pickle)

- send that byte sequence to the interchange over ZMQ
- do a bit of book keeping
- create and return an executor future back to the invoking DFK - this is how we're going to signal to the DFK that the task is completed (with a result or failure) so it is part of the propagation route of results all the way back to the user.

2.3 the interchange

The interchange 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. so whenever a new task arrives from the user workflow process, or when a manager is ready for work, a match is made. there won't always be available work or available workers so there are queues 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. (TODO: what link here? if more stuff merged into Parsl, then the PR can be linkable. otherwise later on maybe a SuperComputing 2024 publication - but still unknown)

so now, the interchange sends the task over one of those two zmq-over-TCP connections I talked about earlier... and we're now on the worker node where we're going to run the task.

2.4 the process worker pool

Generally, a copy of the process worker pool runs on each worker node. (other configurations are possible) and 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)

worker processes - each worker process is a worker. there are a bunch of configuration parameters and algorithms to decide how many workers to run - this happens near the start of the process worker pool process in the manager code. (TODO: link to worker pool code that calculates number of workers)

the task arrives at the manager, and the manager dispatches it to a free worker. it is possible there isnt' a free worker, becuase of the preloading feature for high throughput (TODO link to docstring) - and the task will have to wait in another queue here - but that is a rarely used feature.

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

so at this point, we invoke the function with those arguments (link to the `f(*args, **kwargs)` line)

and the user code runs!

it's 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 we're ignoring that here)

now we've got the task outcome - either a Python object that is the result, or a Python object that is the exception. We pickle that ob-

ject and send it back to the manager, then to the interchange (over the *other* ZMQ-over-TCP socket) and then to the high throughput excecutor submit-side in the user workflow process.

Back on the submit side, there's a high throughput executor process running listening on that socket. It gets the result package and sets the result into the executor future (TODO code reference). That is the mechanism by which the DFK sees that the executor has finished its work, and so that's where the final bit of "task elaboration" (TODO: link to elaboration chapter) happens - the big elaboration here would be retries on failure, which is basically do that whole HTEX submission again and get a new executor future for the next try. (but other less common elaborations would be storing checkpointing info for this task, and file staging)

When that elaboration is finished (and didn't do a retry), we can set that same result value into the AppFuture which all that long time ago was given to the user. And so now future.result() returns that results (or raises that exception), back in the user workflow, and we're done.

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 the task overview, I assumed that process worker pools magically existed on worker nodes. In this section, I'll talk a little bit more about how that actually happens, using Parsl's provider abstraction.

The theme of this section is: get process worker pools running on some nodes that we want to do the work.

We don't need to describe the work (much), because once the workers are running they'll get their own work from the interchange, as I talked about in the previous section.

LRM providers, batch jobs, workers, scaling strategies, batch job environments (esp worker_init)

CHAPTER

FOUR

SERIALIZING TASKS WITH PICKLE AND DILL

TODO: an emphasis on the common parsl problems: (un)installed packages, functions and exceptions

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 *Serializing tasks with Pickle and dill* library (TODO: hyperlink pickle).

As I mentioned in an earlier section, (TODO: backlink hyperlink?) when htex 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."

TODO: review my pickle talk, figure out what is relevant or not. maybe don't need to talk about pickle VM opcodes, just the remote-execution facility at a higher level? and the import facility at a higher level? no need to talk about recursive objects - that's not a user facing problem (unless you're trying to build your own pickle scheme)

4.1 More info

I've talked about Pickle in more depth and outside of the Parsl context at PyCon Lithuania (TODO: link slides and video)

Proxystore - reference its use in Parsl, and reference a citation for just proxystore. TODO

CHAPTER

FIVE

ELABORATING TASKS

stuff that the DFK does to a task that isn't "just run this task"

dependencies, retries, checkpointing, file staging, join_app joining, monitoring resource wrapper

CHAPTER

SIX

UNDERSTANDING THE MONITORING DATABASE

this should focus on making use of data in the monitoring database, not on how monitoring is architected, exceptions

CHAPTER

SEVEN

MODULARITY AND PLUGINS

which bits you can swap for other plugins: how and why

why includes sustainability work on different quality of code/maintenance

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)

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.

Parsl exists as a library within the python ecosystem. Python *exists*.

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

CHAPTER

EIGHT

COLOPHON

Written in rst

Rendered with sphinx

Edited with `vi` and vscode

This text was prepared against Parsl version 2024.08.19