Parsl live-coding speaker notes - PyHEP 2019

http://parsl-project.org (http://parsl-project.org) - "Use Parsl to create parallel programs comprised of Python functions and external components. Execute Parsl programs on any compute resource from laptops to supercomputers."

please interrupt with questions at any point

Me:

I'm not a HEP person. But I do work with one and she's the one that sent me here.

I'm a software engineer - I work both in industry and academia - in addition to Python, my other big language is Haskell, which leads to me being something of a category theorist too...

My driving application for my parsl work at the moment is LSST - the Large Synoptic Survey Telescope under construction in Chile.

Other people here have talks about what parsl gets used for in HEP which is good because that's an area that I'm weak on.

I'm going to do some live coding here to show the basic programming model, and use that as a base for further discussion.

In [85]:

```
# first I did `pip install parsl`

import parsl

parsl.load()
```

In [87]:

```
help(parsl) # or online: https://parsl.readthedocs.io/en/latest/
```

In [10]:

```
import time

def pi_estim_A():
    time.sleep(5)
    return 4
```

In [11]:

```
pi_estim_A()
```

Out[11]:

```
4
```

In [12]:

```
@parsl.python_app
def pi_estim_B():
    time.sleep(5)
    return 4
```

In [17]:

```
future = pi_estim_B()
type(future)
```

Out[17]:

```
parsl.dataflow.futures.AppFuture
```

In [18]:

```
future.result()
```

Out[18]:

```
4
```

In [22]:

```
# so here we get some concurrency: this will take 5 seconds, not 10 seconds...
# after we've launched the first call which returns immediately, we can then go on to do other stuff
# such as launch the second call.
# and then we'll block only when we try to get the result - that first result() call will take 5s,
# but the second call is probably ready already

f1 = pi_estim_B()
f2 = pi_estim_B()
(f1.result() + f2.result())/2
```

Out[22]:

```
4.0
```

In [78]:

```
# now move towards better pi estimation...
# circle inscribed in a square... pick points.

import random
```

In [79]:

```
coords = [(random.random(), random.random()) for _ in range(1,10)]
coords
```

Out[79]:

```
[(0.2635793537336275, 0.9361245514197177),
 (0.8085463877839812, 0.15354898439763187),
 (0.38953131773089333, 0.06710308785772112),
 (0.7171680653097668, 0.7517450975208319),
 (0.40590326562231993, 0.07993332106239615),
 (0.25499432354957996, 0.8290339591243582),
 (0.2167517452560207, 0.0744189763249592),
 (0.4703638006013976, 0.39593162349741584),
 (0.39813327263729836, 0.7063382027058719)]
```

In [38]:

```
@parsl.python_app
def pi_estim_D( coords ):
    time.sleep(2)
    (x,y) = coords
    return 4
```

In [39]:

```
fs = list(map(pi_estim_D, coords))
```

In [49]:

```
# can run this repeatedly and watch the list slowly go from running state to finished state over about 10 seco
nds
fs
```

Out[49]:

```
[<AppFuture super=<AppFuture at 0x7f156c5917b8 state=finished returned int>>,
 <AppFuture super=<AppFuture at 0x7f156c591a20 state=finished returned int>>,
 <AppFuture super=<AppFuture at 0x7f156c591be0 state=finished returned int>>,
 <AppFuture super=<AppFuture at 0x7f156c591da0 state=finished returned int>>,
 <AppFuture super=<AppFuture at 0x7f156c51d080 state=finished returned int>>,
 <AppFuture super=<AppFuture at 0x7f156c51d240 state=finished returned int>>,
 <AppFuture super=<AppFuture at 0x7f156c591160 state=finished returned int>>,
 <AppFuture super=<AppFuture at 0x7f156c5844e0 state=finished returned int>>,
 <AppFuture super=<AppFuture at 0x7f156c584f98 state=finished returned int>>]
```

In [50]:

```
rs = [f.result() for f in fs]
rs
```

Out[50]:

```
[4, 4, 4, 4, 4, 4, 4, 4, 4]
```

In [55]:

```
# now implement circle inside square

import math

@parsl.python_app
def pi_estim_E( coords ):
    (x,y) = coords
    if math.sqrt(x*x + y*y)>=1:
        return 0
    else:
        return 4
```

In [58]:

```
fs = list(map(pi_estim_E, coords))
```

In [61]:

```
rs = [f.result() for f in fs]
rs
```

Out[61]:

```
[4, 4, 4, 4, 4, 0, 4, 0, 4]
```

In [62]:

```
sum(rs) / len(rs)
```

Out[62]:

```
3.111111111111111
```

In [63]:

```
@parsl.python_app
def avg(*args):
    return sum(args)/len(args)
```

In [67]:

```
# i'll pass in the *futures* here, not the results
# and parsl will block until all those futures are done (which they are)
# and then run the code
af = avg(*fs)
type(af)
```

Out[67]:

```
parsl.dataflow.futures.AppFuture
```

In [68]:

```
af.result()
```

Out[68]:

```
3.111111111111111
```

In [69]:

```
# so put this together: we can launch 500 pi_estim_E and the avg all at once, and it will put things into the
# right order but parallelised.
# this is the bit to really explain hard: we get a future out the end, but that future won't get its result
# until all of the first 500 futures have completed and then the avg code runs.
# there's a particular kind of concurrency here which is more constrained than (for example) threads or MPI,
# but (usually) easier to reason about/debug

def estimate_pi():
    coords = [(random.random(), random.random()) for _ in range(1,500)]
    fs = list(map(pi_estim_E, coords))
    return avg(*fs)
```

In [71]:

```
f = estimate_pi()
```

In [72]:

```
f
```

Out[72]:

```
<AppFuture super=<AppFuture at 0x7f156c325860 state=finished returned float>>
```

In [73]:

```
f.result()
```

Out[73]:

```
3.062124248496994
```

In [82]:

```
estimate_pi().result()
```

Out[82]:

```
3.094188376753507
```

In [ ]:

```

```

In [ ]:

```

```

this above is all on my laptop at the start, ran parsl.load(). i could instead pass in a configuration here that describes how to run on other systems - that can describe how to connect to that machine, how to submit a batch job, where the working directories are, ... above calls would be unchanged but we'd get remote execution on a cluster.

support classic batch systems like Slurm, torque, cobalt... also more cloud-like stuff like kubernetes clusters or Amazon web services.

can also launch @bash_apps which are shell commands - that's pretty common to launch tools that are already packaged as a CLI rather than a python library - an example is some astronomy simulation running on 2000 nodes, running jobs that run for 12 or more hours on and off for a number of months

someone allegedly has run a million tasks in one parsl run.

parsl can also manage those decorated calls in other ways that aren't just parallelisation:

- other locations as talked about above
- checkpointing/memoization -- it looks like you're making a function call but parsl remembers you've made that already (in this run, or in your set of checkpoints) and returns the value from that cache.
- retries - if an invocation fails, can retry it - instead of getting a failure out of result() after that first failure, parsl can try a few times - that won't fix your code if that is what is broken, but it can help survive transient failures -eg queue time running out
- can describe input and output files on disk, and parsl can copy them around
- working on monitoring - collecting both task and node information as we run, and then make it available in different ways: visualisation, in an sql db, through API calls

There's also work we've done with packaging up environments for your remote execution:

- containers (eg kubernetes, singularity)
- Cooperative Computing Tools group at Notre Dame - package up local conda environment

There are a number of APIs for plugging in different kinds of backends to support different systems below parsl, if you want to plug in the back, not write workflow stuff on the front: support different execution platforms, file transfer systems, batch systems, ...

Community: couple of weeks ago we had ParslFest in Chicago! about 36 people there. Pretty broad range of science areas that people are using parsl - this is not a specific HEP tool. http://parsl-project.org/parsl_meeting (http://parsl-project.org/parsl_meeting)

We're on github https://github.com/Parsl/parsl (https://github.com/Parsl/parsl) and there is a slack (invite on the main website) with a #parsl-help channel.

Also, we don't support python2.

Also, stickers!