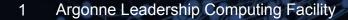
# Enabling Workflows with Parsl

Antonio Villarreal, Argonne Leadership Computing Facility





### **Goals of this Demo**

- Explain the Basic Parsl Functionality
- Demonstrate Parsl Wrappers on Python Definitions
- Demonstrate Parsl Wrappers to Make Bash Calls
- Demonstrate Parsl Submitting Jobs to Theta
- Experiment with Job Monitoring Utilities



#### **How Does Parsl Work?**

- Parsl is a Python library (<u>https://parsl.readthedocs.io/</u>)
- Parsl augments Python with decorator functions
- Parsl can connect these decorated functions via Python objects or files
- Parsl runs a work manager that then executes these tasks scaling to thousands of compute nodes



#### How to Install Parsl on Theta

Installing Parsl is relatively straightforward on Theta!

- 1. module load miniconda-3
- 2. pip install --user parsl
- 3. optional: pip install --user parsl[monitoring]

With this, you are ready to run Parsl! It will require Python 3.5+.



#### **A Simple Test Case**

For this demo, we will be doing a very simple algorithm—calculating the value of pi via a Monte Carlo approach. In brief:

- 1. We sample a number of random points between 0 and 1 in two dimensions.
- 2. We determine if these points lie within a quarter circle of radius 1.
- 3. By the ratio of the number of points within the circle to those outside of the circle, we can compute pi.



#### pi\_test\_local.py

For our first example, we look at only a local execution. The following code block defines an **executor**, which determines how the Parsl manager will run tasks.

#### from parsl.executors import ThreadPoolExecutor

```
parsl_config = Config(
    executors=[ThreadPoolExecutor(
        max_threads=8,
        label='login-node'
        )
    ],
    strategy=None,
)
parsl.load(parsl_config)
```

ThreadPoolExecutor will use local compute threads. We've limited it to 8 threads and attached a label to this executor for later.



### pi\_test\_local.py

We now decorate our python function to estimate pi with the @python\_app decorator.

# from parsl.app.app import python\_app

```
@python_app(executors=['login-node'])
def estimate_pi(n_points):
    import numpy as np
    x = np.random.uniform(0,1,n_points)
    y = np.random.uniform(0,1,n_points)
    dist = np.sqrt(x*x+y*y)
    n_circle = np.sum(dist <= 1)
    pi_est = 4*n_circle/n_points
    return pi_est</pre>
```

Here we refer back to the executor label from earlier. By default, this will use any available executor, rather than specific resources.



## pi\_test\_local.py

We can then run this by calling: python pi\_test\_local.py
Each time estimate pi is called, Parsl will

define a new task to be run on its workers. The output becomes a **future**, which can then be passed to other functions.

To get our results, we use the .result() method on each task and then average them together!

```
if __name__ == '__main__':
    import numpy as np
    n_points = 100000
    n_trials = 100
    trials = []
    for i in range(n_trials):
        trials.append(estimate_pi(n_points))
    outputs = [i.result() for i in trials]
    print(np.mean(outputs))
```



### pi\_test\_bash.py

#### from parsl.app.app import bash\_app

@bash\_app(executors=['login-node'])
def calc\_pi(num\_points, outputs=[]):
 return 'python /home/antoniov/repos/parsl-pi-test/calc\_pi.py {} &> {}'.format(num\_points, outputs[0])

We can also execute arbitrary bash commands via the @bash\_app decorator.

Note: this decorator does not return a python object and will require you to string commands together using output/input files as necessary. The return string is executed in bash.

Handy for calling non-Python code or using containers!

See pi\_test\_bash.py and the README for more details.



### pi\_test\_queue.py

#### The

HighThroughputExecutor allows you to run thousands of Parsl workers simultaneously.

It will also submit a job request to Theta using Cobalt, potentially even spacing workers across multiple job submissions.

```
parsl_config = Config(
    executors=[
        HighThroughputExecutor(
            label='theta-htex',
            max_workers = WORKERS_PER_NODE*MY_COMPUTE_NODES*MY_COMPUTE_BLOCKS,
            worker_debug=True,
            address_address_by_hostname(),
            provider=CobaltProvider(
                queue=MY_QUEUE,
                account=MY_ACCOUNT,
                launcher=AprunLauncher(overrides="-d 64"),
                walltime=MY_TIME,
                nodes_per_block=MY_COMPUTE_NODES,
                init_blocks=1,
                min_blocks=1,
                max_blocks=MY_COMPUTE_BLOCKS,
                # string to prepend to #COBALT blocks in the submit
                # script to the scheduler eg: '#COBALT -t 50'
                scheduler_options='',
                # Command to be run before starting a worker, such as:
                worker_init='module load miniconda-3; export PATH=$PATH:{}'.format(MY_USER_PATH),
                cmd_timeout=120,
```



#### pi\_test\_queue.py

The provider keyword lets us define all elements necessary for job submission.

- 1. queue and account contain information about your Theta allocation.
- 2. walltime and nodes\_per\_block set your job submission parameters.
- 3. init\_blocks refers to the number of initial job submissions.
- 4. max\_blocks refers to how many job submissions to keep at once.
- 5. launcher defines how Parsl workers should launch tasks.
- 6. worker\_init appends a bash command ahead of launching workers on resources.
  - a. The included export helps with finding a worker specific task.

You may also define a max number of workers to keep up simultaneously.



After changing MY\_USER\_PATH to include your account name, you can run this task with:

python pi\_test\_queue.py

You may want to do this in a screen, as the parsl driver will be now running, waiting for the allocated resources to become available!



### pi\_test\_monitoring.py

Parsl can also run a sqlite monitoring database which keeps additional info!

# from parsl.addresses import address\_by\_hostname from parsl.monitoring.monitoring import MonitoringHub

The monitoring keyword is added onto the Parsl configuration and creates a database named monitoring.db in the directory of the Parsl driver.

Details on all the stored tables can be located in the README.md.

monitoring=MonitoringHub(
 hub\_address=address\_by\_hostname(),
 hub\_port=55055,
 monitoring\_debug=False,
 resource\_monitoring\_interval=10,



### **Thanks for Listening**

Some helpful links!

- Parsl Documentation: <u>https://parsl.readthedocs.io/en/stable/</u>
- Parsl Tutorials: <u>https://github.com/Parsl/parsl-tutorial</u>

If you have any other questions, don't hesitate to reach out to Antonio Villarreal on Slack!

