# Parsl: Enabling Scalable Interactive Computing in Python

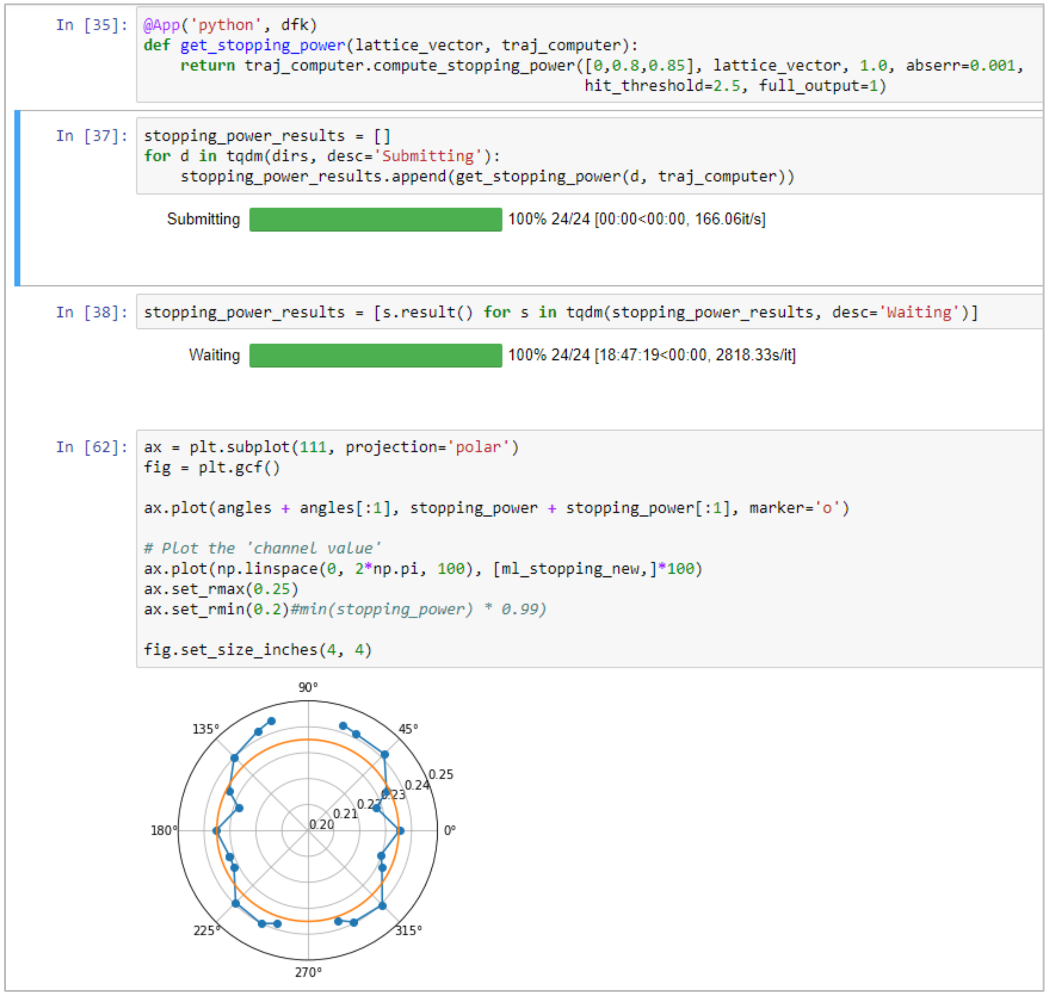
Kyle Chard (chard@uchicago.edu)

Yadu Babuji, Anna Woodard, Ben Clifford, Lukasz Lacinski, Mike Wilde, Dan Katz, Justin Wozniak, Ian Foster

http://parsl-project.org

# An increasingly common story…

- I'm developing an application and I need to link together external tools + functions
  - (where each tool is dependent on data from the previous tool)
- I have a notebook that does *X* and I need to run it on a cloud, cluster, supercomputer
- I need to run my analysis using a range of local and distributed datasets
- …
- And I want to do this in an interactive environment



```
In [35]: @App('python', dfk)
         def get_stopping_power(lattice_vector, traj_computer):
             return traj_computer.compute_stopping_power([0,0.8,0.85], lattice_vector, 1.0, abserr=0.001,
                                                          hit_threshold=2.5, full_output=1)

In [37]: stopping_power_results = []
         for d in tqdm(dirs, desc='Submitting'):
             stopping_power_results.append(get_stopping_power(d, traj_computer))

         Submitting 100% 24/24 [00:00<00:00, 166.06it/s]

In [38]: stopping_power_results = [s.result() for s in tqdm(stopping_power_results, desc='Waiting')]

         Waiting 100% 24/24 [18:47:19<00:00, 2818.33s/it]

In [62]: ax = plt.subplot(111, projection='polar')
         fig = plt.gcf()

         ax.plot(angles + angles[:1], stopping_power + stopping_power[:1], marker='o')

         # Plot the 'channel value'
         ax.plot(np.linspace(0, 2*np.pi, 100), [ml_stopping_new,]*100)
         ax.set_rmax(0.25)
         ax.set_rmin(0.2)#min(stopping_power) * 0.99)

         fig.set_size_inches(4, 4)
```

# Parsl: Interactive parallel scripting in Python

Annotate functions to make Parsl *apps*

- Python apps call Python functions
- Bash apps call external applications

Apps return “futures”: a proxy for a result that might not yet be available

Apps run concurrently respecting data dependencies.
Natural parallel programming!

Parsl scripts are independent of where they run. Write once run anywhere!

```
pip install parsl
```

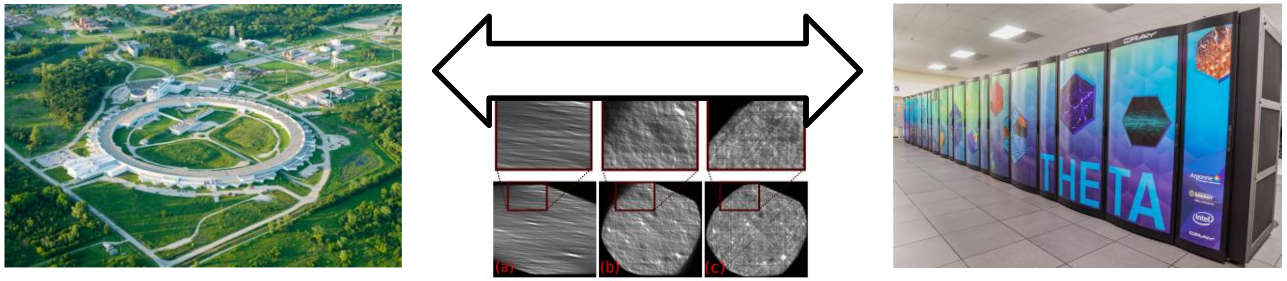
```
@python_app
def hello ():
    return 'Hello World!'

print(hello().result())
```

```
Hello World!
```

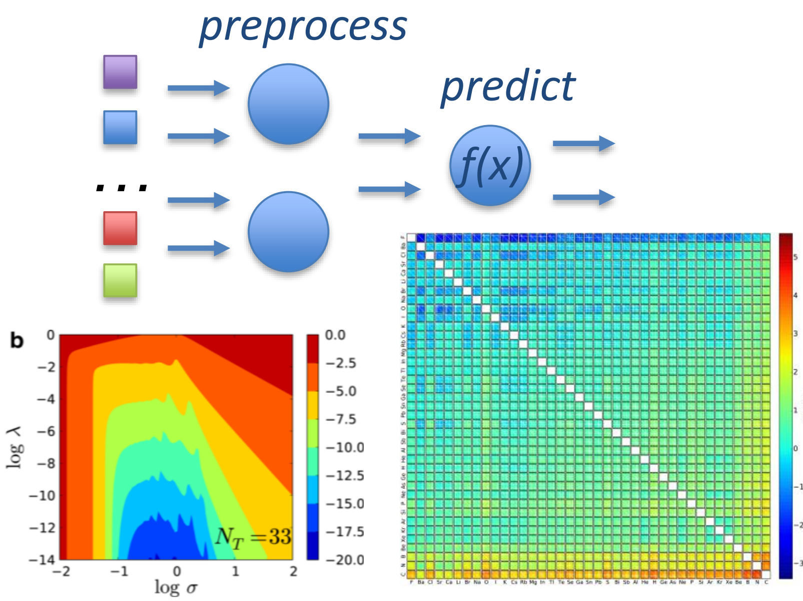
```
@bash_app
def echo_hello(stdout='echo-hello.stdout'):
    return 'echo "Hello World!"'

echo_hello().result()

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

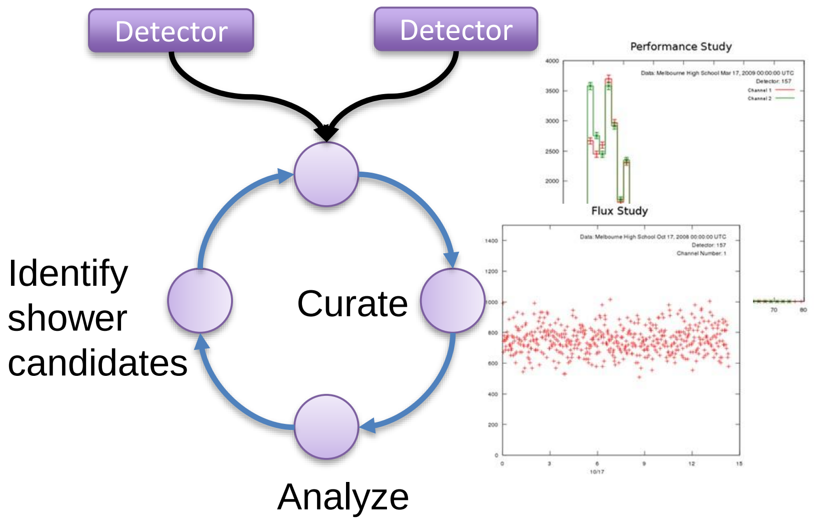
```
Hello World!
```

# Scientific workflows are not just for MTC/HPC/HTC



Online computing





Machine learning

Interactive computing

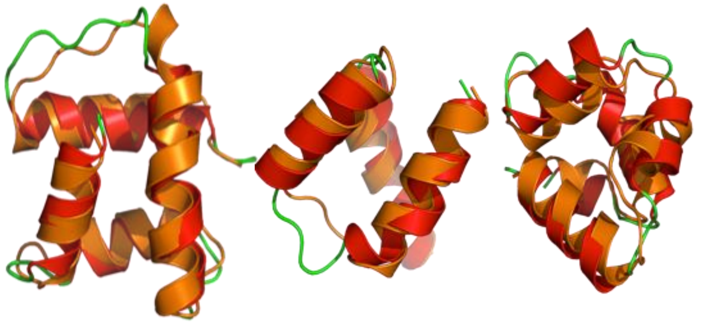
# When do you need automated workflow?

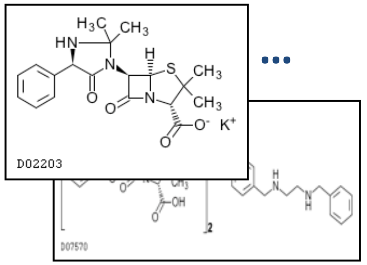
## Example application: protein-ligand docking for drug screening

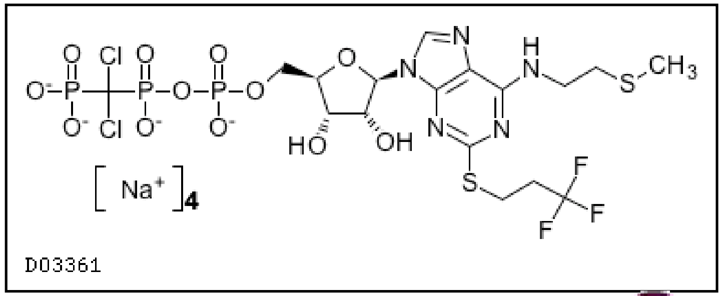
*O(10) proteins implicated in a disease*

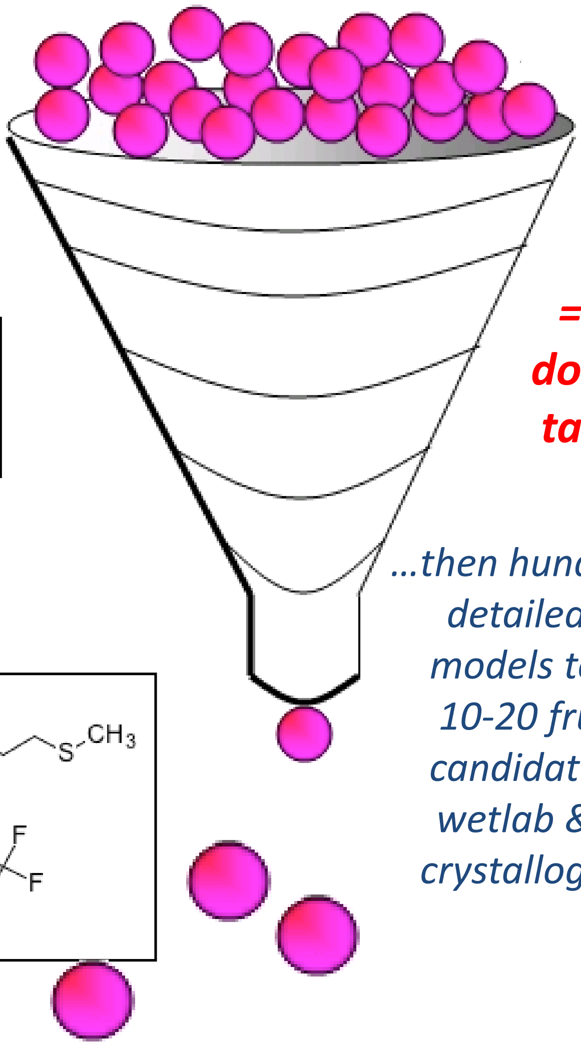
**X**

*O(100K) drug candidates*









*= 1M docking tasks…*

*…then hundreds of detailed MD models to find 10-20 fruitful candidates for wetlab & APS crystallography*

## Expressing a many task workflow in Parsl

*1) Wrap the protein docking code:*

```
@bash_app
def dock(p, c, minRad, maxRad)
  return 'dock.sh {0} {1} {2} {3}'.format(p,
   c ,minRad, maxRad)
```

## Expressing a many task workflow in Parsl

*2) Execute the protein docking workflow:*

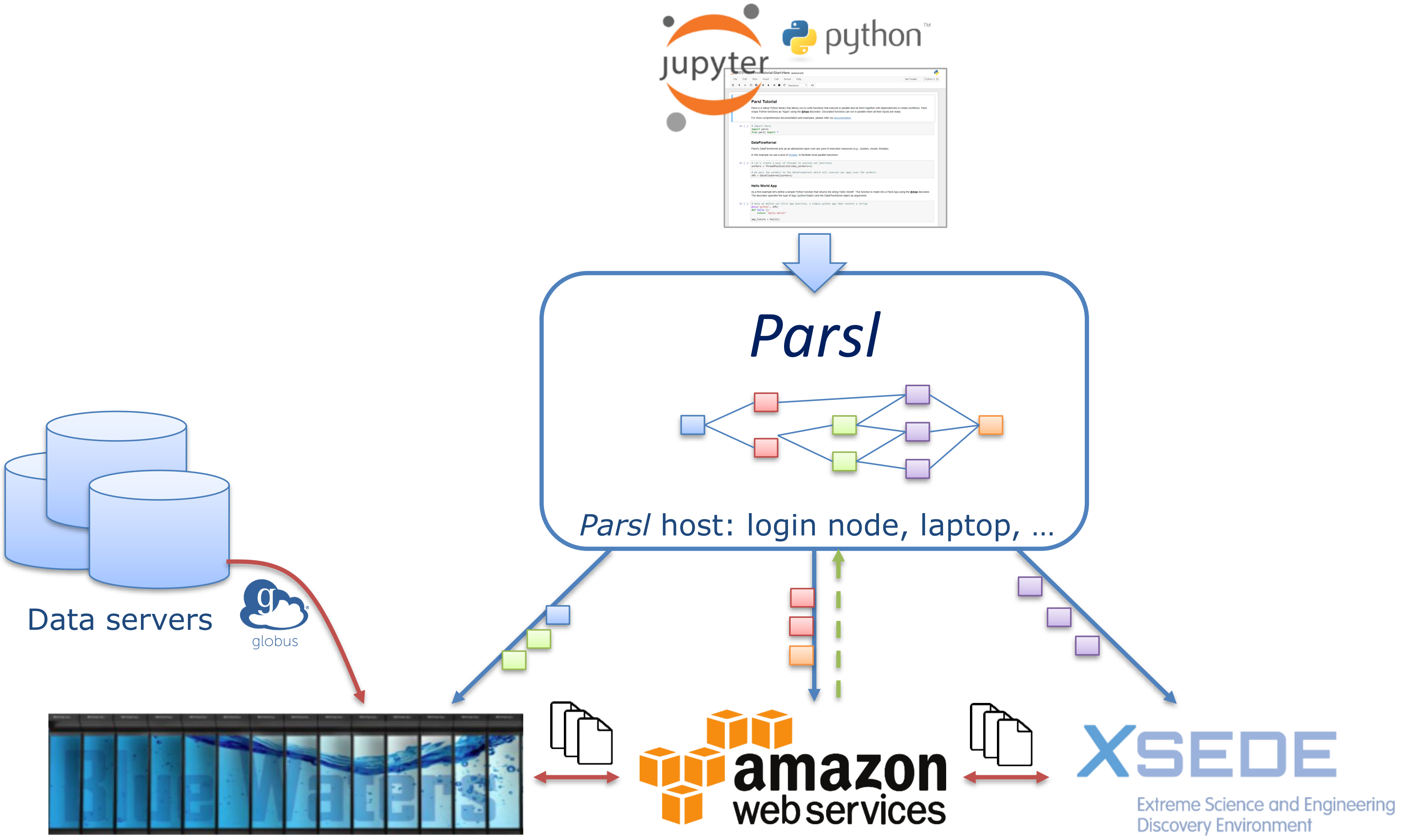
```
for p in proteins:
   for c in ligands:
      structure[p][c] =
       dock(p, c, minRad, maxRad)

scatter_plot = analyze(structure)
```

# Brief history: the Swift parallel scripting language

- 10+ years of development
- C-like language with implicit parallelism
- Applied in dozens of scientific domains
- Data management, multi-site execution, coasters, etc.
- We are leveraging lessons and components to build Parsl

```
type file;

app (file o) simulation (int sim_steps, int sim_range, int sim_values)
{
  simulate "--timesteps" sim_steps "--range" sim_range "--nvalues" sim_values
stdout=filename(o);
}

app (file o) analyze (file s[])
{
  stats filenames(s) stdout=filename(o);
}

int nsim   = toInt(arg("nsim","10"));
int steps  = toInt(arg("steps","1"));
int range  = toInt(arg("range","100"));
int values = toInt(arg("values","5"));

file sims[];

foreach i in [0:nsim-1] {
  file simout <single_file_mapper; file=strcat("output/sim_",i,".out")>;
  simout = simulation(steps,range,values);
  sims[i] = simout;
}

file stats<"output/average.out">;
stats = analyze(sims);
```

## Parsl in action: dynamic dataflow execution


jupyter
python™
Parsl
Parsl host: login node, laptop, …
Data servers
globus
amazon web services
XSEDE
Extreme Science and Engineering Discovery Environment
File Edit View Insert Cell Kernel Help
Not Trusted
Python 3
Markdown

First-Tutorial-Start-Here (autosaved)

**Parsl Tutorial**

Parsl is a native Python library that allows you to write functions that execute in parallel and tie them together with dependencies to create workflows. Parsl wraps Python functions as "Apps" using the **@App** decorator. Decorated functions can run in parallel when all their inputs are ready.

For more comprehensive documentation and examples, please refer our documentation

```
In [ ]: # Import Parsl
        import parsl
        from parsl import *
```

**DataFlowKernal**

Parsl's DataFlowKernel acts as an abstraction layer over any pool of execution resources (e.g., clusters, clouds, threads).

In this example we use a pool of threads, to facilitate local parallel exectuion.

```
In [ ]: # Let's create a pool of threads to execute our functions
        workers = ThreadPoolExecutor(max_workers=4)

        # We pass the workers to the DataFlowKernel which will execute our Apps over the workers.
        dfk = DataFlowKernel(workers)
```

**Hello World App**

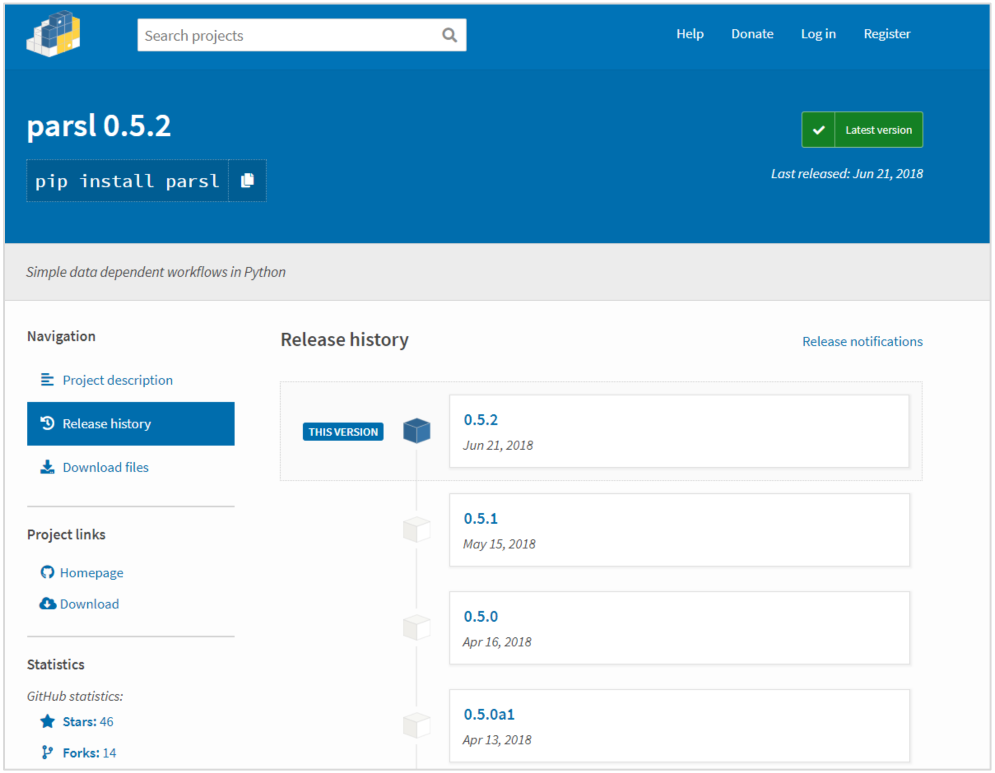
As a first example let's define a simple Python function that returns the string 'Hello World!'. This function is made into a Parsl App using the **@App** decorator. The decorator specifies the type of App ('python'|'bash') and the DataFlowKernel object as arguments.

```
In [ ]: # Here we define our first App function, a simple python app that returns a string
        @App('python', dfk)
        def hello ():
            return 'Hello World!'

        app_future = hello()
```

## Parsl is Python

```
pip3 install parsl
```

- Use Python libraries natively
- Stage Python data transparently
- Integrates with Python ecosystem



parsl 0.5.2

```
pip install parsl
```

Latest version

Last released: Jun 21, 2018

Simple data dependent workflows in Python

Navigation

- Project description
- Release history
- Download files

Project links

- Homepage
- Download

Statistics

GitHub statistics:

- Stars: 46
- Forks: 14

Release history

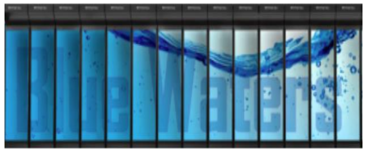
Release notifications

THIS VERSION

| Version | Date |
|---|---|
| 0.5.2 | Jun 21, 2018 |
| 0.5.1 | May 15, 2018 |
| 0.5.0 | Apr 16, 2018 |
| 0.5.0a1 | Apr 13, 2018 |

## Parsl scripts are execution provider independent

- The same script can be run locally, on grids, clouds, or supercomputers
  - Works directly with the scheduler (no HTC-like setup)
- Containers can be used for per-app execution or repeated invocation of the same app
- Parsl builds on libsubmit
  - https://github.com/Parsl/libsubmit
- Currently supported execution providers:
  - Local, Cloud (AWS, Azure, private), Slurm, Torque, Condor, Cobalt



amazon web services

XSEDE

Google Cloud

Open Science Grid

parsl

# Separation of code and execution

```python
@python_app(executors=['midway'])
def midway():
    return 'I am run on midway!'

@bash_app(executors=['local'])
def local():
    return 'I am run locally!'
```

```python
from libsubmit.channels import SSHChannel
from libsubmit.providers import SlurmProvider

import parsl
from parsl.config import Config
from parsl.executors.ipp import IPyParallelExecutor
from parsl.executors.threads import ThreadPoolExecutor

config = Config(
    executors=[
        IPyParallelExecutor(
            label='midway',
            provider=SlurmProvider(
                'westmere',
                channel=SSHChannel(
                    hostname='swift.rcc.uchicago.edu',
                    username='annawoodard'
                ),
                max_blocks=1000,
                nodes_per_block=1,
                tasks_per_node=6,
                overrides='module load singularity; module load Anaconda3/5.1.0; source activate parsl_py36'
            ),
        ),
       ThreadPoolExecutor(label='local', max_threads=2)
    ],
)

parsl.load(config)
```

Pilot jobs on a cluster

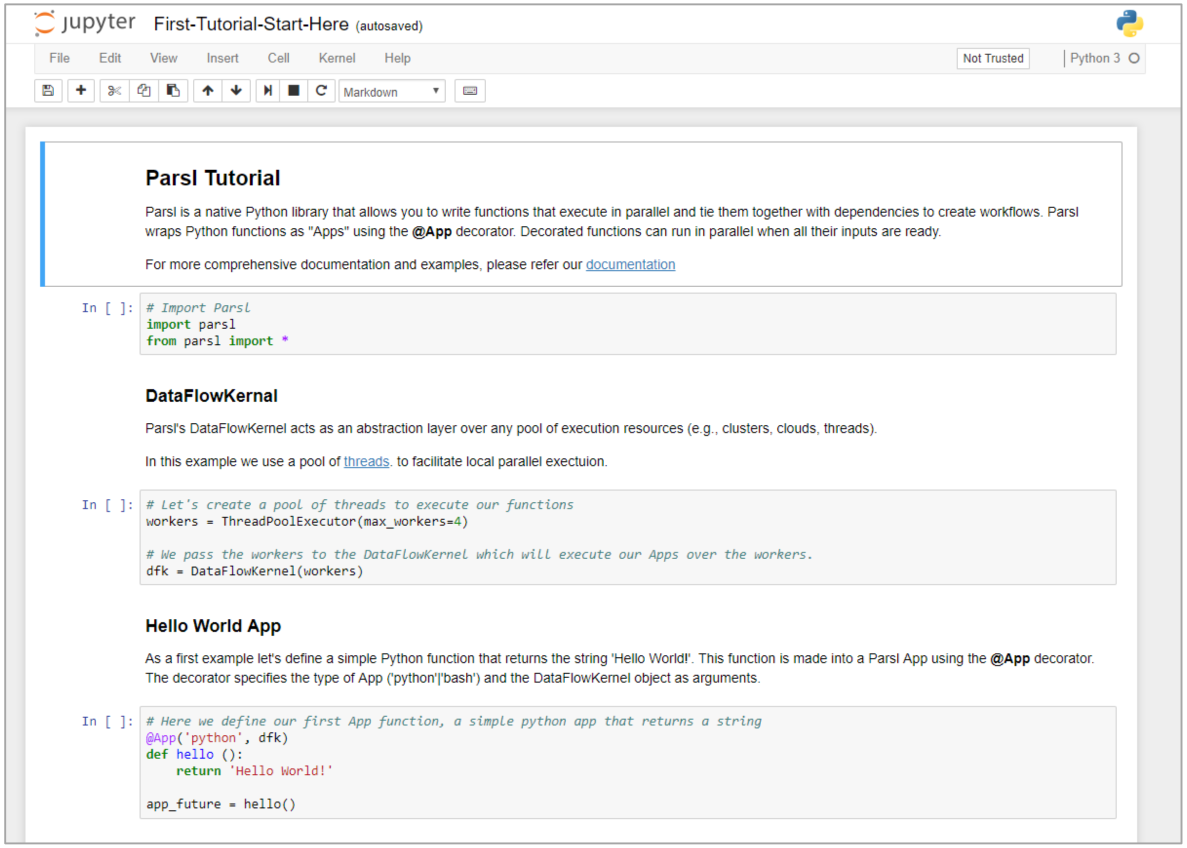
Local threads

* Config format for Parsl 0.6

parsl

# Interactive supercomputing in Jupyter notebooks

- Parsl can be used within a Jupyter notebook with no modifications necessary
- Tunneling and OAuth-based flows supports remote execution from the notebook
- Visualization of Parsl graph in notebook


jupyter First-Tutorial-Start-Here (autosaved)
File Edit View Insert Cell Kernel Help
Not Trusted
Python 3
Markdown

### Parsl Tutorial

Parsl is a native Python library that allows you to write functions that execute in parallel and tie them together with dependencies to create workflows. Parsl wraps Python functions as "Apps" using the **@App** decorator. Decorated functions can run in parallel when all their inputs are ready.

For more comprehensive documentation and examples, please refer our documentation

```
In [ ]: # Import Parsl
        import parsl
        from parsl import *
```

**DataFlowKernal**

Parsl's DataFlowKernel acts as an abstraction layer over any pool of execution resources (e.g., clusters, clouds, threads).

In this example we use a pool of threads. to facilitate local parallel exectuion.

```
In [ ]: # Let's create a pool of threads to execute our functions
        workers = ThreadPoolExecutor(max_workers=4)

        # We pass the workers to the DataFlowKernel which will execute our Apps over the workers.
        dfk = DataFlowKernel(workers)
```

**Hello World App**

As a first example let's define a simple Python function that returns the string 'Hello World!'. This function is made into a Parsl App using the **@App** decorator. The decorator specifies the type of App ('python'|'bash') and the DataFlowKernel object as arguments.

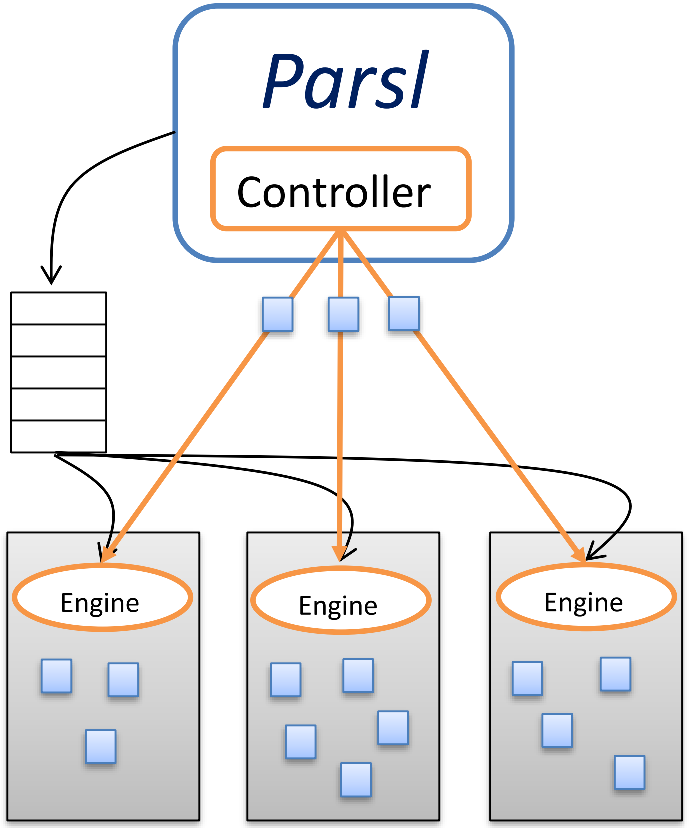
```
In [ ]: # Here we define our first App function, a simple python app that returns a string
        @App('python', dfk)
        def hello ():
            return 'Hello World!'

        app_future = hello()
```

parsl

## Parsl supports a variety of execution models

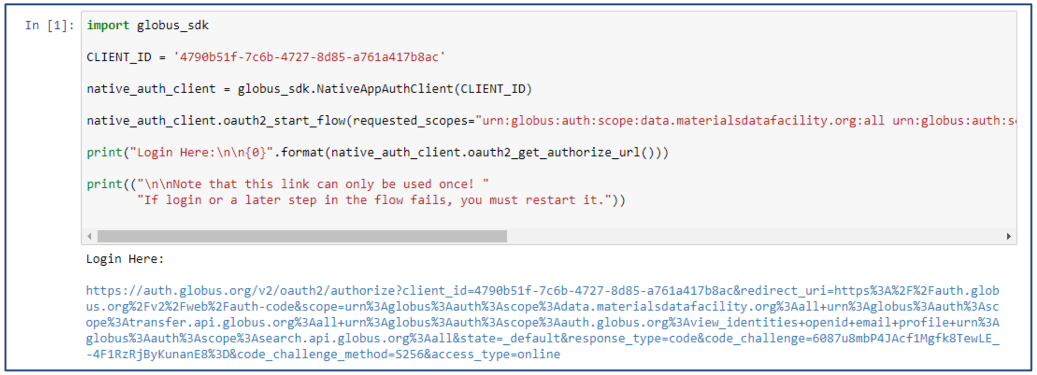
- Threads
  - Local execution
- Ipython.parallel
  - Pilot job model
- Swift/T
  - Extreme scale execution

- New execution models can be added

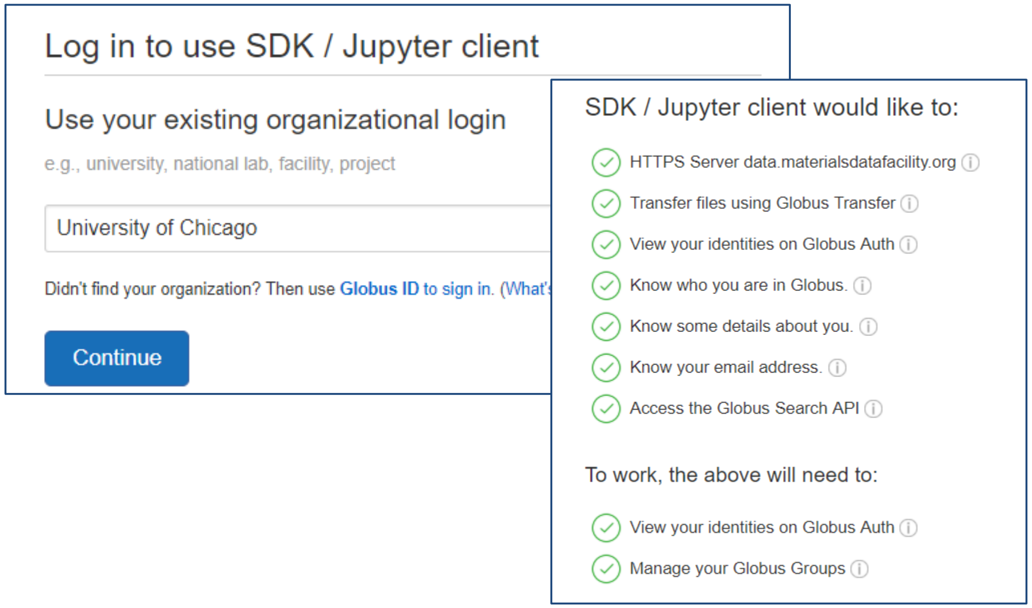


# Authentication and authorization

- A&A is hard today
  - 2FA, X509, etc.
- Integration with Globus Auth to support native app integration for accessing Globus (and other) services
- Using scoped access tokens, refresh tokens, delegation support



```
In [1]: import globus_sdk

        CLIENT_ID = '4790b51f-7c6b-4727-8d85-a761a417b8ac'

        native_auth_client = globus_sdk.NativeAppAuthClient(CLIENT_ID)

        native_auth_client.oauth2_start_flow(requested_scopes="urn:globus:auth:scope:data.materialsdatafacility.org:all urn:globus:auth:s

        print("Login Here:\n\n{0}".format(native_auth_client.oauth2_get_authorize_url()))

        print(("\n\nNote that this link can only be used once! "
               "If login or a later step in the flow fails, you must restart it."))

        Login Here:

        https://auth.globus.org/v2/oauth2/authorize?client_id=4790b51f-7c6b-4727-8d85-a761a417b8ac&redirect_uri=https%3A%2F%2Fauth.glob
        us.org%2Fv2%2Fweb%2Fauth-code&scope=urn%3Aglobus%3Aauth%3Ascope%3Adata.materialsdatafacility.org%3Aall+urn%3Aglobus%3Aauth%3Asc
        ope%3Atransfer.api.globus.org%3Aall+urn%3Aglobus%3Aauth%3Ascope%3Aauth.globus.org%3Aview_identities+openid+email+profile+urn%3A
        globus%3Aauth%3Ascope%3Asearch.api.globus.org%3Aall&state=_default&response_type=code&code_challenge=6087u8mbP4JAcf1Mgfk8TewLE_
        -4F1RzRjByKunanE8%3D&code_challenge_method=S256&access_type=online
```



Log in to use SDK / Jupyter client

Use your existing organizational login

e.g., university, national lab, facility, project

University of Chicago

Didn't find your organization? Then use Globus ID to sign in. (What's

SDK / Jupyter client would like to:

- ✓ HTTPS Server data.materialsdatafacility.org
- ✓ Transfer files using Globus Transfer
- ✓ View your identities on Globus Auth
- ✓ Know who you are in Globus.
- ✓ Know some details about you.
- ✓ Know your email address.
- ✓ Access the Globus Search API

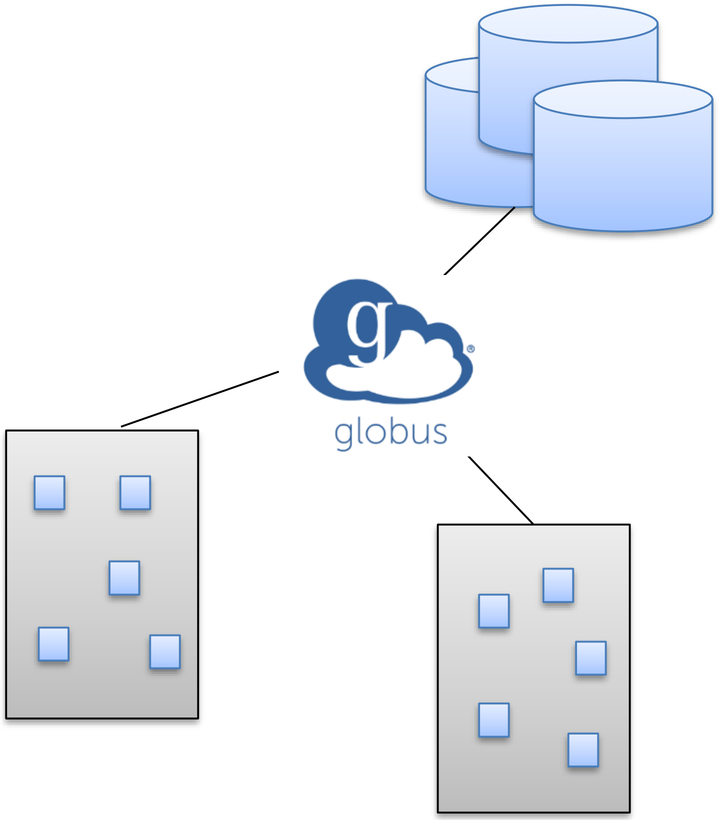
To work, the above will need to:

- ✓ View your identities on Globus Auth
- ✓ Manage your Globus Groups

## Parsl provides transparent (wide area) data management

- Implicit data movement to/from repositories, laptops, supercomputers, …
- Globus for third-party, high performance and reliable data transfer
  - Support for site-specific DTNs
- HTTP/FTP direct data download/upload
- Compliments node-specific staging and caching models

```
parsl_file =
   File(globus://EP/path/file)
```

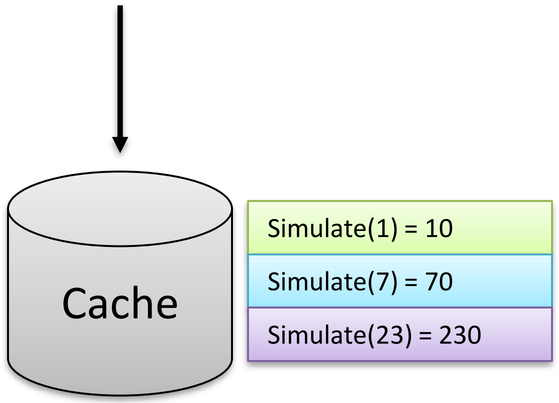


www.globus.org

# App caching (memoization)

- Parsl apps are often expensive to recompute
- In many development modes results need not be recomputed
  - During development or interactive workflow
- Memoization optimizes execution by caching app results when called with the same inputs
- Parsl relies on user control to annotate deterministic functions

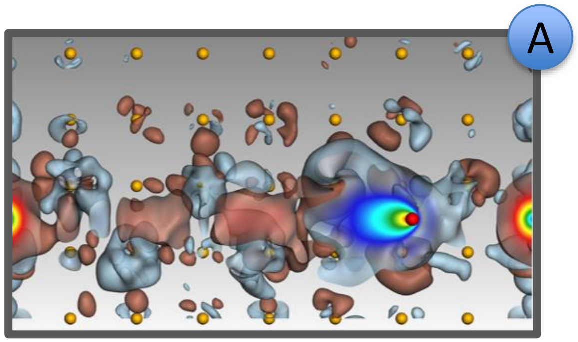
```
@python_app(cache=True)
def simulate(input_variable):
      return input_variable * 10
```

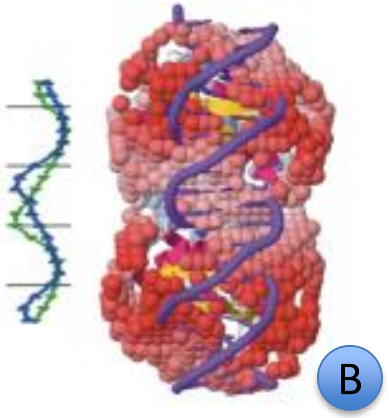


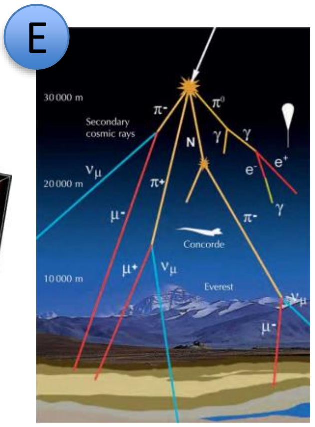
parsl

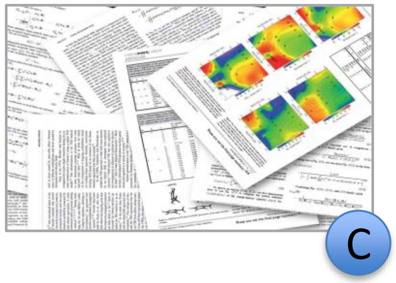
# Scientific applications using Parsl

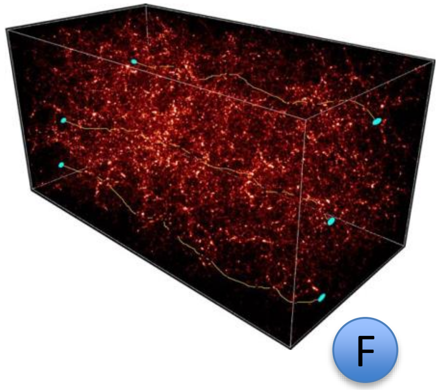
- A Machine learning to predict stopping power in materials
- B Protein and biomolecule structure and interaction
- C Information extraction to discovery facts in publications
- D Materials science at the Advanced Photon Source
- E Cosmic ray showers as part of QuarkNet
- F Weak lensing using sky surveys
- G Machine learning and data analytics (DLHub)

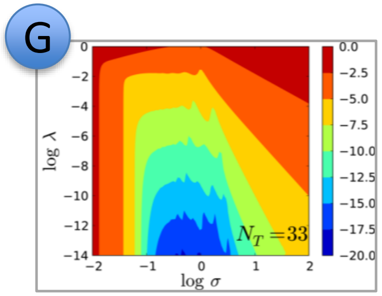


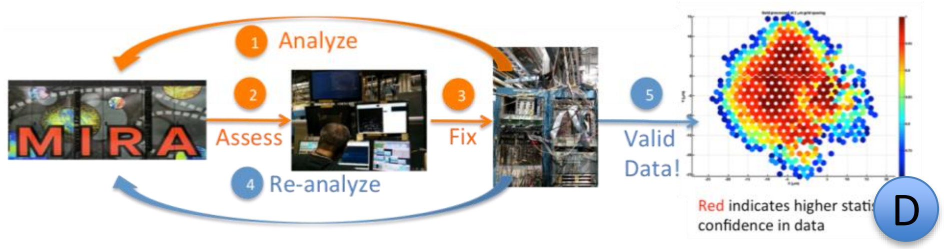












## Summary

- Parsl's implicit dataflow model in Python allows for simple expression of complex dependencies
  - Expressed directly in Python
  - Can be used to implement a range of workflow models
- Parsl integrates with the scientific ecosystem
  - Development and execution of scalable applications in Jupyter
  - Use of common SciPy libraries
  - Integration with Globus
- In Parsl, code is separate from the specification of computing resources and data location: this makes Parsl scripts portable and scalable
- Parsl has a number of other important features:
  - app caching, checkpointing, elasticity, container support, data transfer, and more

# Questions?

**http://parsl-project.org**

Try Parsl: http://try.parsl-project.org

https://mybinder.org/v2/gh/Parsl/parsl-tutorial/master