

Parsl: A Parallel Scripting Library for Python

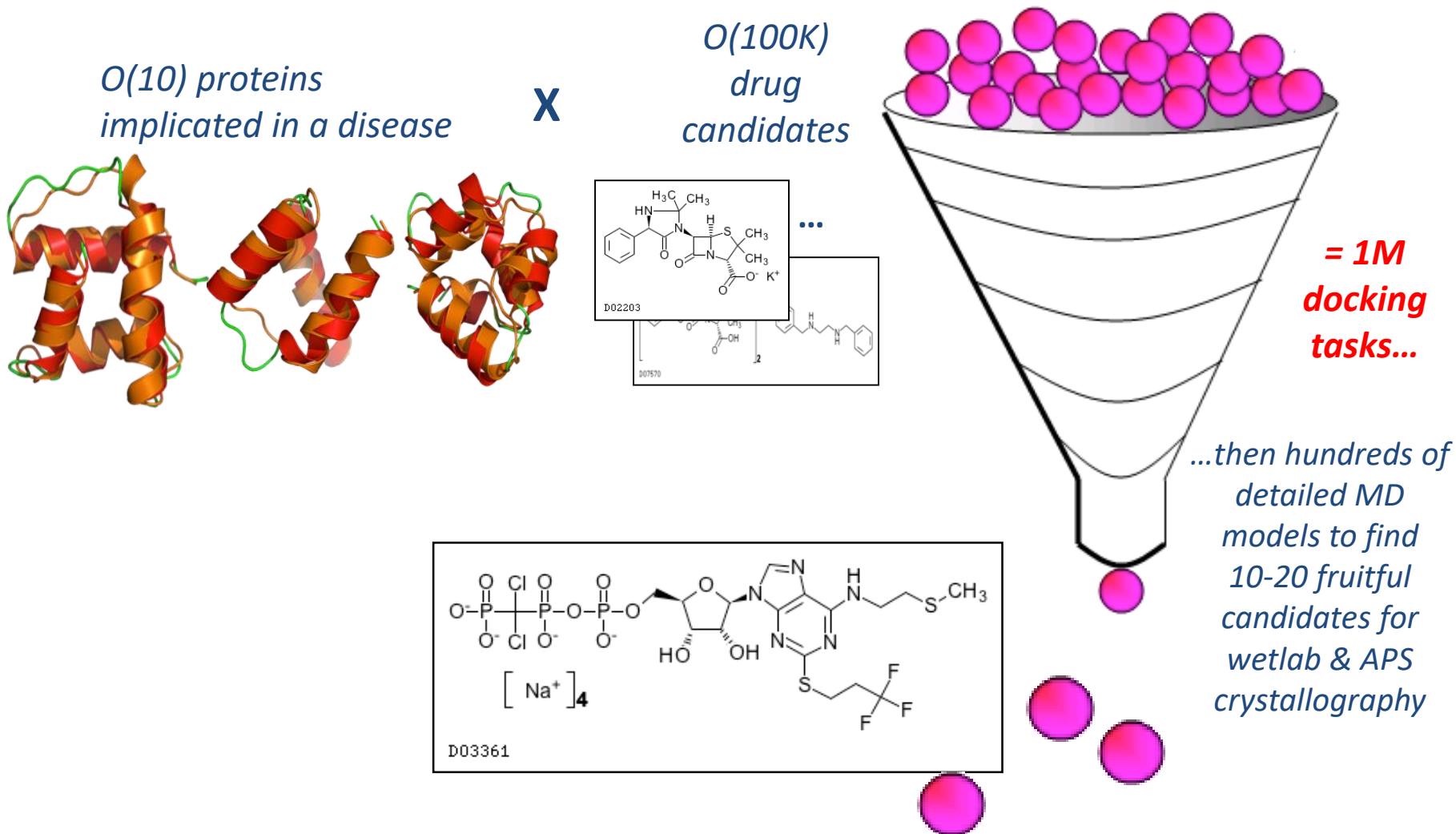
Kyle Chard (chard@uchicago.edu)

Yadu Babuji, Mike Wilde, Dan Katz, Anna Woodard, Justin Wozniak, Ian Foster

<http://parsl-project.org>

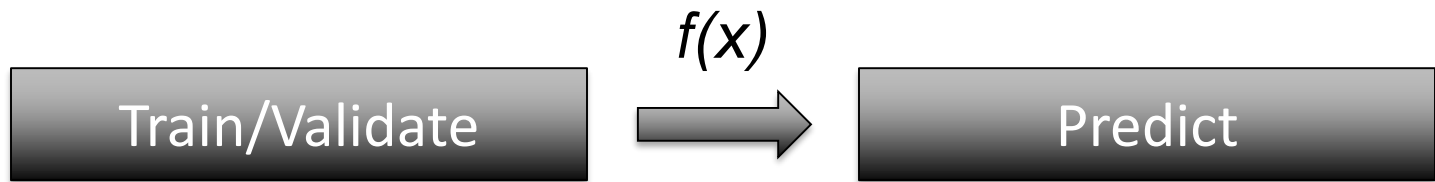
When do you need automated workflow?

Example application: protein-ligand docking for drug screening

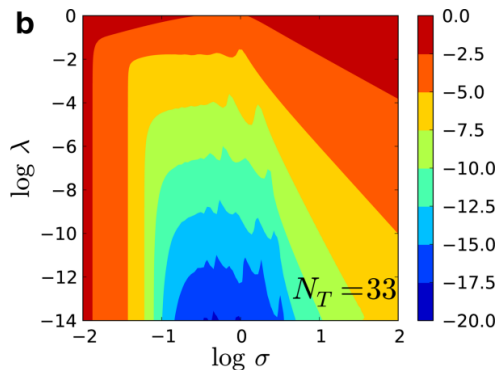
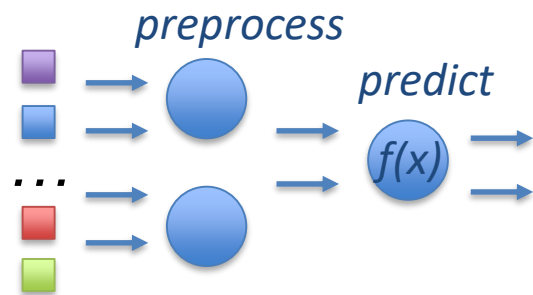
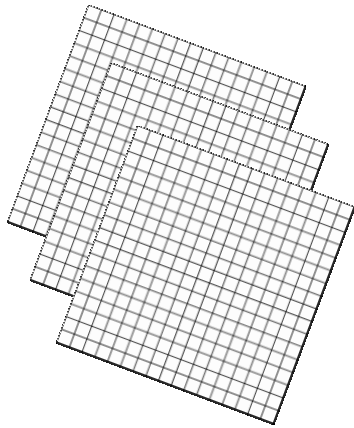


When do you need automated Workflow in machine learning?

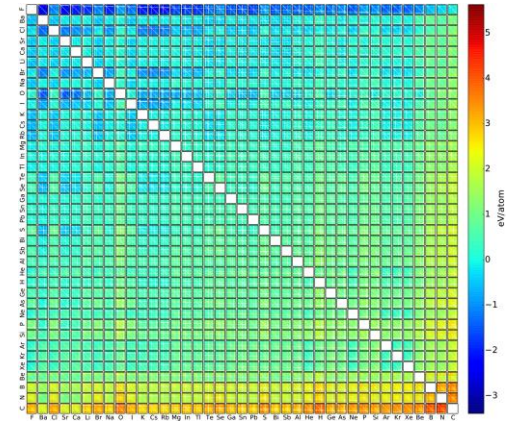
Example application: predicting material design



O(Ms) of data used to train model



O(2M) evaluation of possible designs

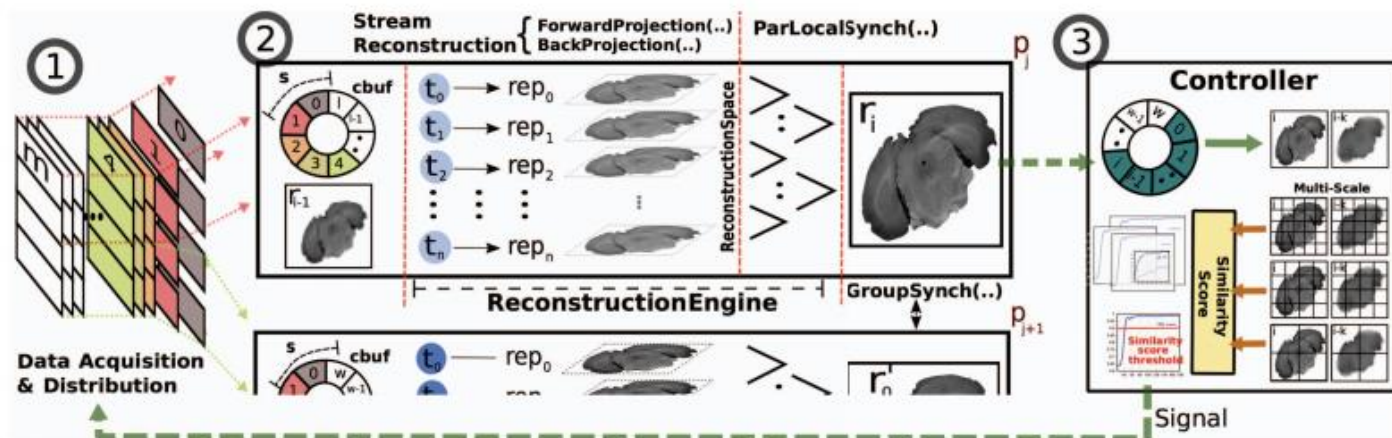
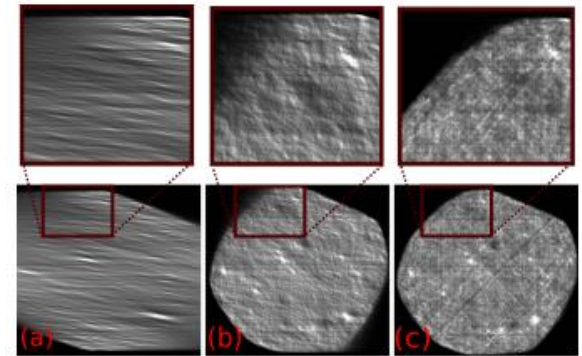
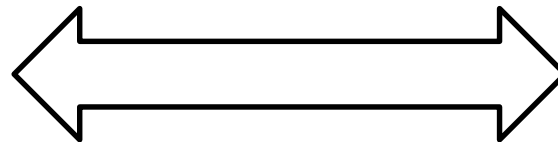


When do you need automated workflow in online experiments?

Example application: cement hardening experiments

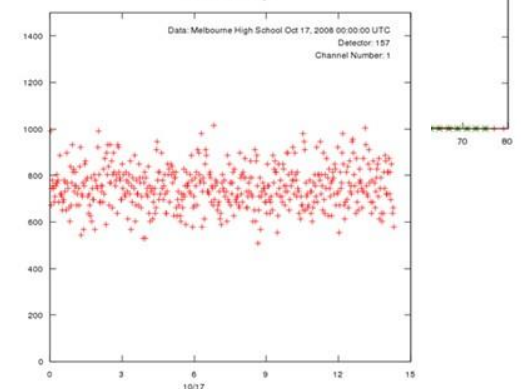
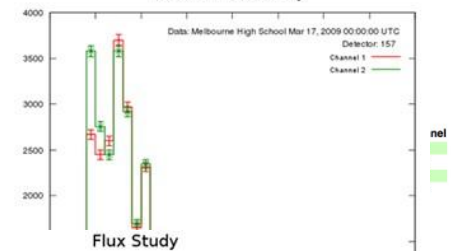
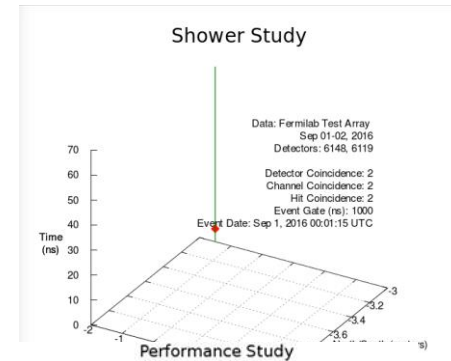
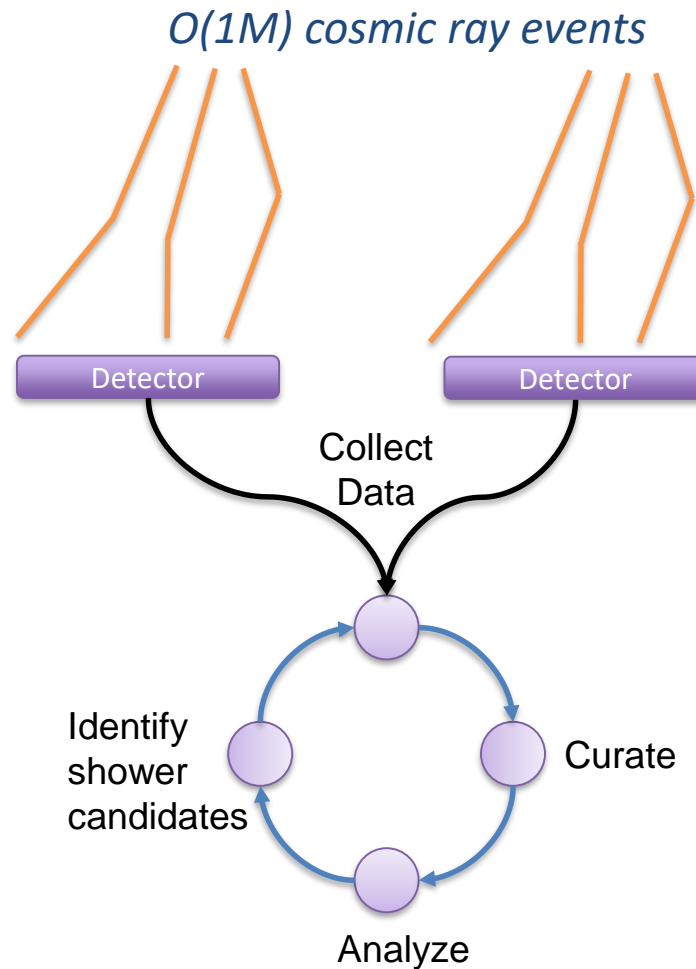
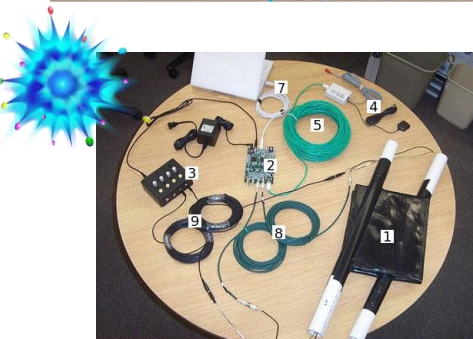
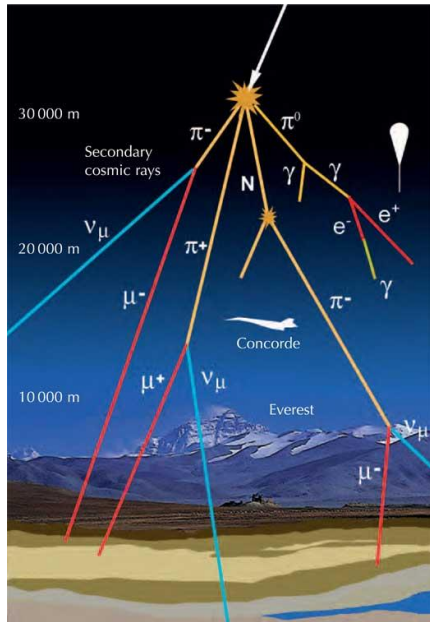


$O(Ms)$ of images per day
(2,000 projections per second, 2,048 x 2,048 pixels)



When do you need automated workflow in interactive science?

Example application: QuarkNet cosmic ray eLab



Workflow requirements

- Usability and ease of workflow expression
- Ability to leverage complex architecture of HPC and HTC systems (fabric, scheduler, hybrid node/programming models), individually and collectively
- Ability to integrate high-performance data services and volumes
- Make use of the system task rate capabilities from clusters to extreme-scale

- Parsl: A Python programming library for programming at any scale

Expressing a many task workflow in Parsl

1) Wrap the protein docking code:

```
@App('bash', dfk)
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)
```


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[];

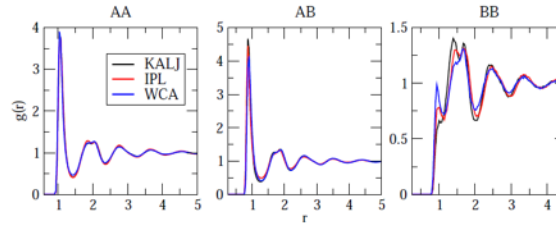
foreach i in [0:nsim-1] {
    file simout <single_file_mapper; file=strcat("output/sim_", i, ".out");
    simout = simulation(steps, range, values);
    sims[i] = simout;
}

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

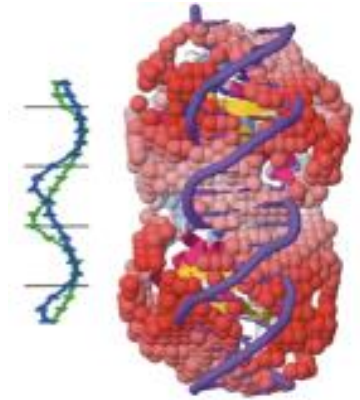
Large-scale applications using Swift

- A** Simulation of super-cooled glass materials
- B** Protein and biomolecule structure and interaction
- C** Climate model analysis and decision making for global food production & supply
- D** Materials science at the Advanced Photon Source
- E** Multiscale subsurface flow modeling
- F** Modeling of power grid for OE applications

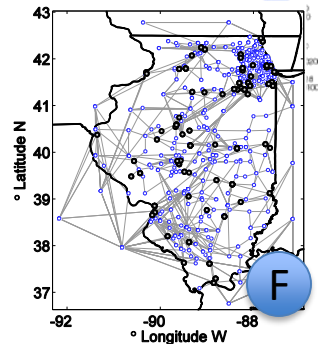
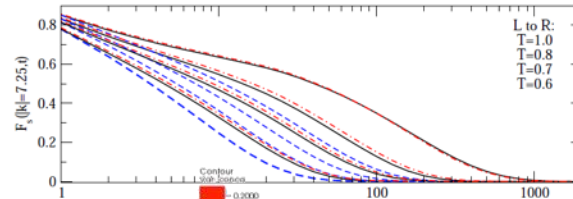
All have published science results obtained using Swift



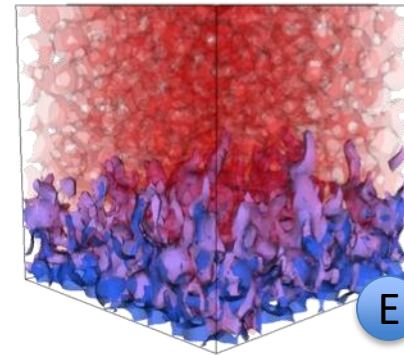
A



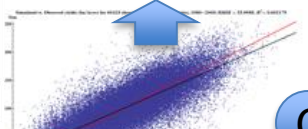
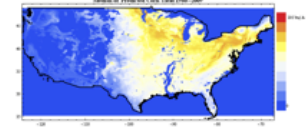
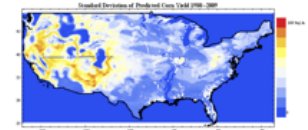
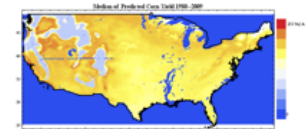
B



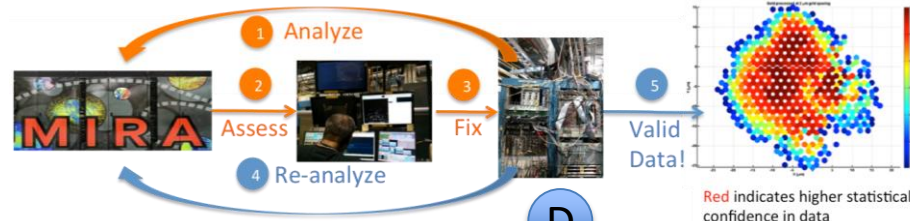
F



E



C



D

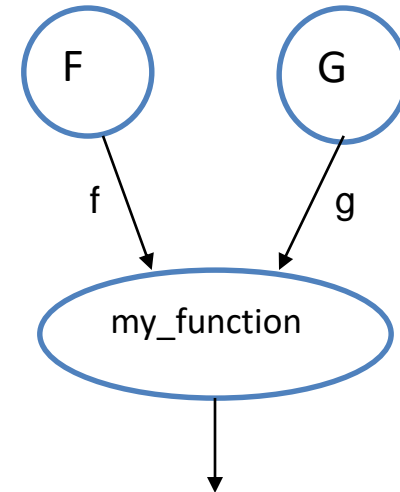
Pervasively parallel

- Parsl is a parallel scripting system for grids, clouds and clusters

```
def my_function(f, g):  
    return f + g
```

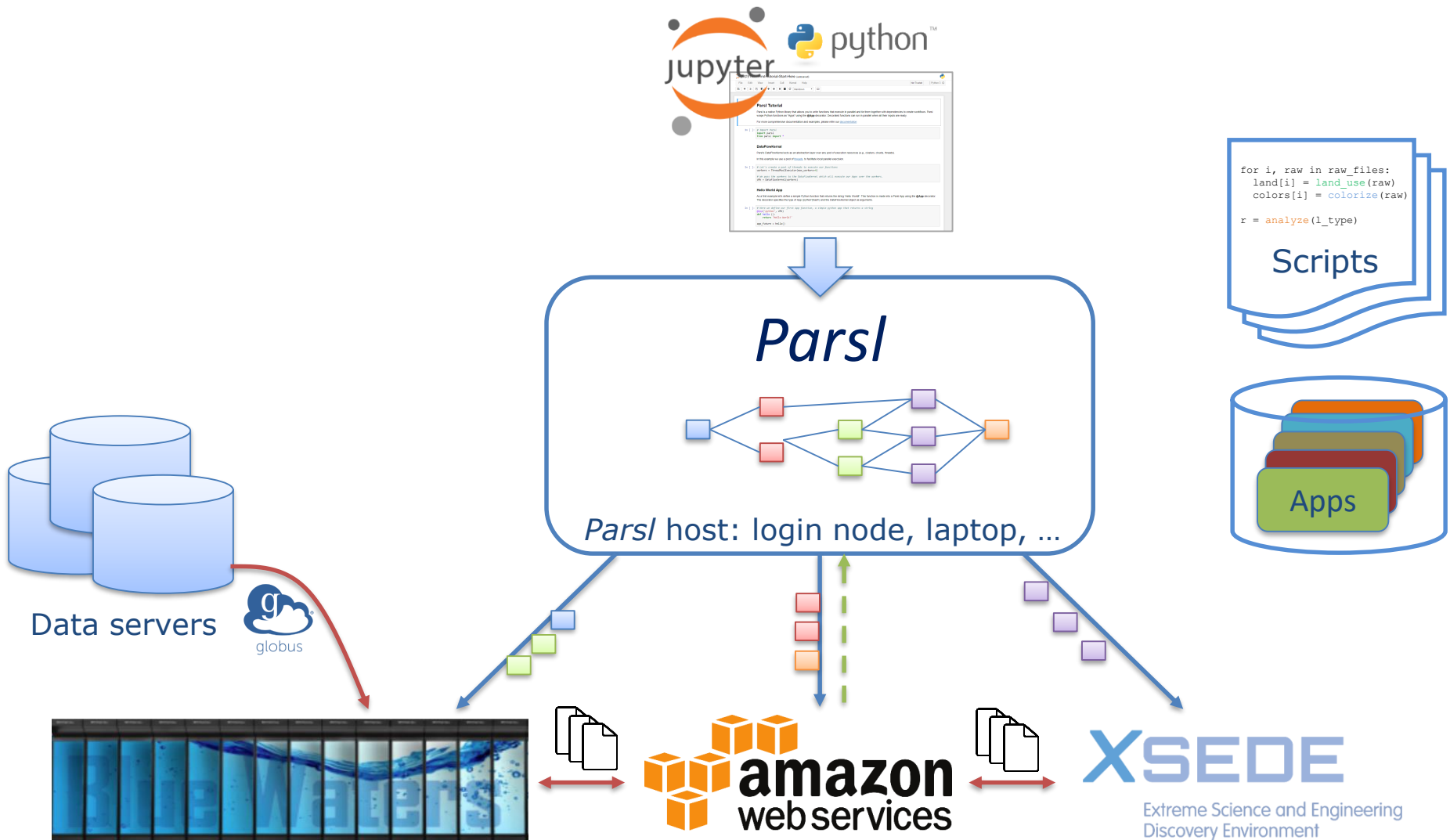
```
f = F(i)  
g = G(i)
```

```
my_function(f, g)
```



- F() and G() are computed in parallel
 - Can be Python functions, or leaf tasks (command line executables or scripts in shell, python, R, Octave, MATLAB, ...)
- App parallelism is *automatic*
- Works recursively throughout the program's call graph

Dynamic dataflow execution



Parsl is Python

```
pip3 install parsl
```

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

The screenshot shows the PyPI page for the 'parsl' package, version 0.3.1. The page includes a search bar, a navigation menu, and a detailed description of the package. The package is described as a 'Simple and easy parallel workflows system for Python'. The page also lists the author, home page, download URL, keywords, license, and categories. A table shows the available files for download, including a Python Wheel and a Source file. The page also includes a 'Downloads' button and a 'Not Logged In' section with links for Login, Register, and Login with OpenID or Google.

python™

» Package Index > parsl > 0.3.1

PACKAGE INDEX »

- Browse packages
- List trove classifiers
- RSS (latest 40 updates)
- RSS (newest 40 packages)
- Terms of Service
- PyPI Tutorial
- PyPI Security
- PyPI Support
- PyPI Bug Reports
- PyPI Discussion
- PyPI Developer Info

ABOUT »

NEWS »

DOCUMENTATION »

DOWNLOAD »

COMMUNITY »

FOUNDATION »

CORE DEVELOPMENT »

parsl 0.3.1

Simple data dependent workflows in Python

Downloads ↓

Simple and easy parallel workflows system for Python

Not Logged In

- Login
- Register
- Lost Login?
- Login with OpenID
- Login with Google

Status

Nothing to report

File	Type	Py Version	Uploaded on	Size
parsl-0.3.1-py3-none-any.whl (md5)	Python Wheel	py3	2017-12-03	32KB
parsl-0.3.1.tar.gz (md5)	Source		2017-12-03	23KB

Author: Yadu Nand Babuji
Home Page: <https://github.com/Parsl/parsl>
Download URL: <https://github.com/Parsl/parsl/archive/0.2.1.tar.gz>
Keywords: Workflows, Scientific computing
License: Apache 2.0

Categories

- Development Status :: 3 - Alpha
- Intended Audience :: Developers
- License :: OSI Approved :: Apache Software License
- Programming Language :: Python :: 3.5
- Programming Language :: Python :: 3.6

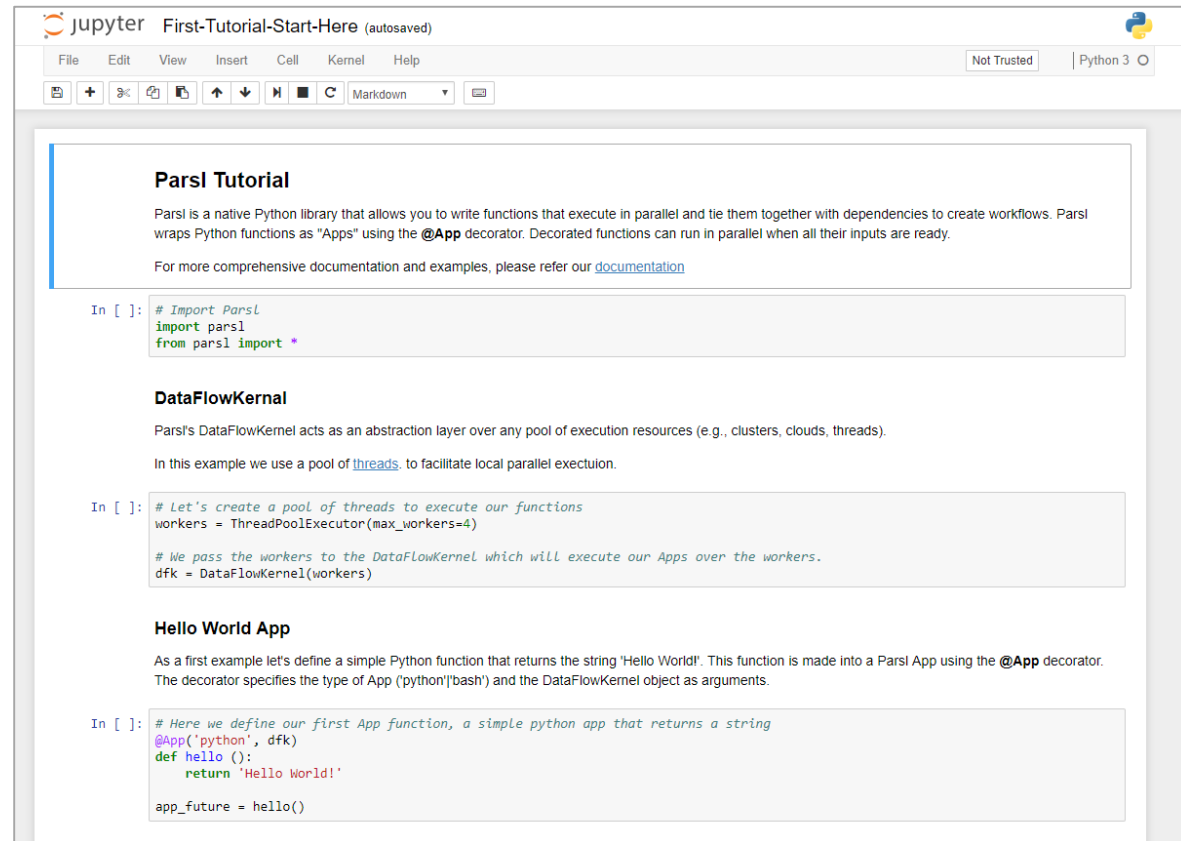
Requires Distributions

- libsubmit (≥=0.2.3)
- ipyparallel

Package Index Owner: yadunand
DOAP record: [parsl-0.3.1.xml](#)

Interactive supercomputing with Jupyter notebooks

- Parsl can be used from within a Jupyter notebook
- Visualization of Parsl graph in notebook
- Soon: transparent pass through of authentication tokens from JupyterHub
- Investigating support for JupyterLab



```
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 *

DataFlowKernel
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 execution.

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)

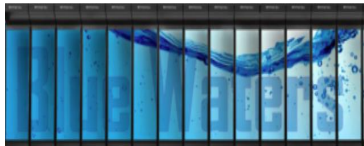
Hello World App
As a first example let's define a simple Python function that returns the string 'Hello World!'. This function is made into a Parsl App using the @App decorator. The decorator specifies the type of App ('python'|'bash') and the DataFlowKernel object as arguments.

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()
```

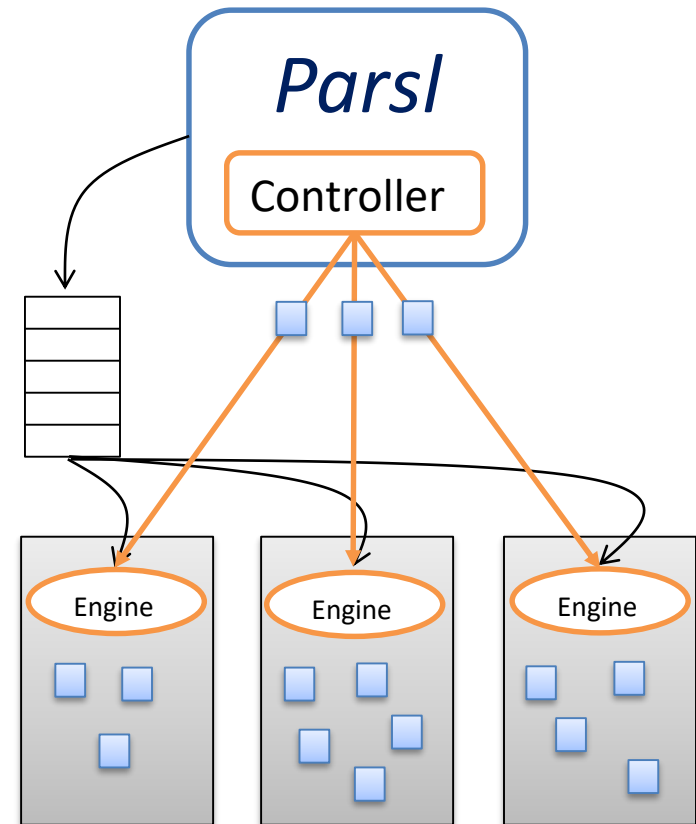
Parsl scripts are execution provider independent

- The same script can be run locally, on grids, clouds, or supercomputers
 - Works directly with the scheduler (no HTC-like setup)
- A single script may use many execution providers
- Parsl builds on libsubmit
 - <https://github.com/Parsl/libsubmit>
- Currently supported execution providers:
 - Local, Cloud (AWS, Azure, private), Slurm, Torque, Condor, Cobalt



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



Multiple sites supported within a single script

- Common for apps to require different execution resources and environments
- Parsl apps may specify the site(s) on which they can be executed
 - Including remote and local execution

```
@App('bash', dfk, sites=["Midway_SB"])
def tleap(input_file, inputs=[], outputs=[], stdout=None, stderr=None, mock=False ):
    return '''module load amber/16; tleap -f %s''' % input_file

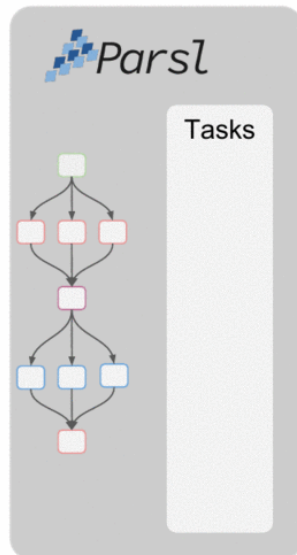
@App('bash', dfk, sites=["Midway_GPU"])
def pmemd_cuda(input_file=None, inputs=[], outputs=[], stdout=None, stderr=None, ref=True ):
    if ref:
        r = "-ref {inputs[1]}"
    else:
        r = ""

    return '''/software/amber-16-el6-x86_64+cuda-8.0/bin/pmemd.cuda -O -i %s -p {inputs[0]}'''
```



Elasticity

- Parsl DAGs are dynamic and grow over time
 - Results in variable workloads with variable resource requirements
- Parsl provides a user-oriented flow control model that:
 - Monitors waiting workload
 - Provisions resources within user-defined bounds according to a parallelism parameter

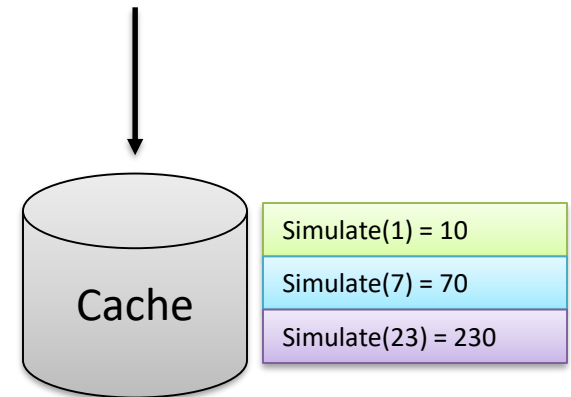


Initializing sites

App caching (memoization)

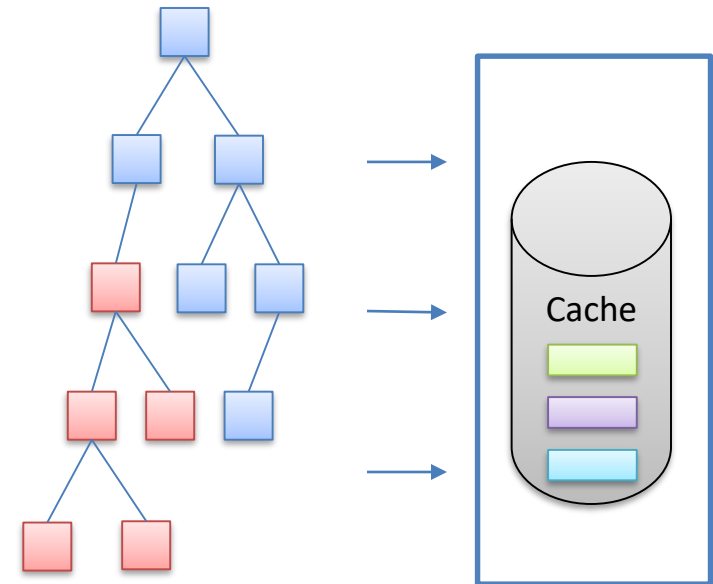
- Parsl apps are often expensive to recompute
- In many development modes results need not be recomputed
 - During development or interactive workflow
- Memoization optimizes execution by caching app results when called with the same inputs
- Parsl relies on user control to annotate deterministic functions

```
@app('Python', dfk, cache=True)  
def simulate(input_variable):  
    return input_variable * 10
```

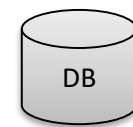


Fault tolerance and checkpointing

- Failure modes:
 - App failure, data error, node failure, etc.
- Lazy vs immediate failure
- Simplest model: workflow level checkpointing
 - Exploits app caching to save app results and reuse results for subsequent execution
 - Python objects and files
- Extensible model for checkpointing to different persistent stores
- Enables automatic retry of workflows and re-execution from saved state



Checkpointing interface



Globus-based authentication and authorization

- A&A is hard today
 - 2FA, X509, etc.
- Building native app integration to provide embedded access to Globus (and other) services
- Using scoped access tokens, refresh tokens, delegation support
- Developing support for (semi-transparent) SSH-based authentication to compute resources

```
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{}".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.globus.org%2Fv2%2Fweb%2Fauth-code&scope=urn%3Aglobus%3Aauth%3Ascope%3Adata.materialsdatafacility.org%3Aall+urn%3Aglobus%3Aauth%3Ascope%3Atransfer.api.globus.org%3Aall+urn%3Aglobus%3Aauth%3Ascope%3Adata.materialsdatafacility.org%3Aall+urn%3Aglobus%3Aauth%3Ascope%3Asearch.api.globus.org%3Aall&state=_default&response_type=code&code_challenge=6887u8mbP4JAcf1Mgfk8TewLE-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

Didn't find your organization? Then use [Globus ID to sign in.](#) (What?!) [Learn more](#)

[Continue](#)

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 ⓘ

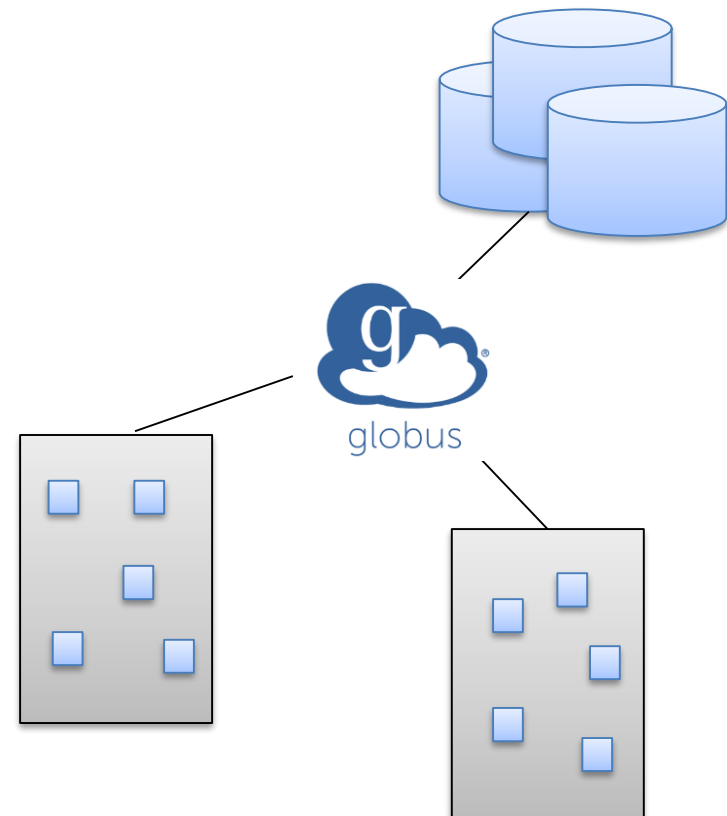
To work, the above will need to:

- ✔ View your identities on Globus Auth ⓘ
- ✔ Manage your Globus Groups ⓘ

Globus data management

- Adding support for transparent high performance and reliable data movement to/from repositories, laptops, supercomputers, ...
 - Initially Globus, then HTTP
- Support for site-specific DTNs
- Compliments node-specific staging and caching models

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

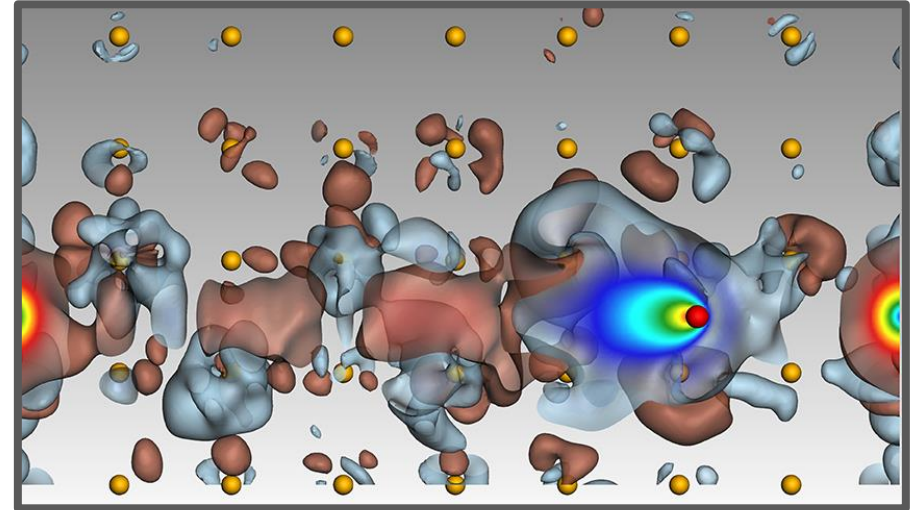


Parsl in action: materials science

Stopping Power: a “drag” force experienced by high speed protons, electrons, or positrons in a material

Areas of Application

- Nuclear reactor safety
- Magnetic confinement / inertial containment for nuclear fusion
- Solar cell surface adsorption
- Medicine (e.g., proton therapy cancer treatment)
- **Critical to understanding material radiation damage**



André Schleife and Cheng-Wei Lee (UIUC)
2016 ALCF INCITE Project
“Electronic Response to Particle Radiation in Condensed Matter”

Computing Stopping Power with TD-DFT

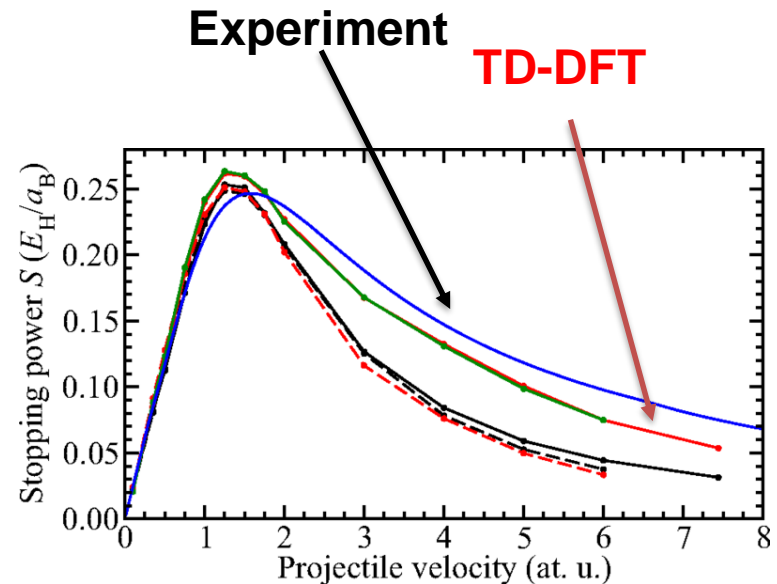
Stopping power (SP) can be accurately calculated by time-dependent density functional theory (TD-DFT)

- ✓ Excellent agreement with experiment
- ✓ Can vary orientation, projectile, material
- ✓ Highly parallelizable

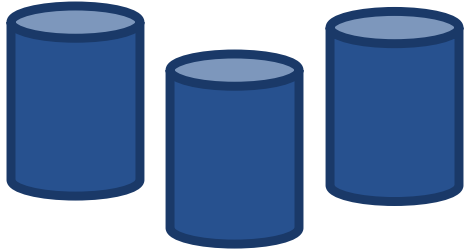
But we need many results

- Direction dependence
- Effect of defects
- Many more materials

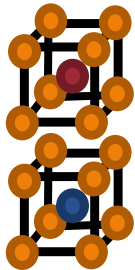
TD-DFT is costly, so use ML too



Parsl enables straightforward parallelization



MATERIALS
DATA
FACILITY



$$\Delta H_f = -1.0$$

$$\Delta H_f = -0.5$$



```
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')]

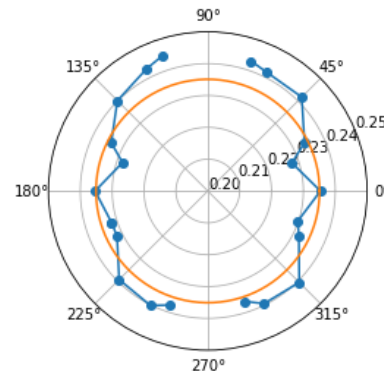
Waiting ██████████ 100% 24/24 [18:47:19<00:00, 2818.33s/it]
```

```
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)
```



LSST: Dark energy weak lensing

- Large Synoptic Survey Telescope (LSST): 10-year sky survey delivering 200 PB of production images
- Small to large scale workflows needed to
 - 1) process images and
 - 2) perform analyses of processed images
- Weak lensing analyses can determine the structure of dark matter, measure the expansion rate of the universe, etc.



LSST: Dark energy weak lensing

```
@App("bash", dfk)
def doTomoBinning (d_des, f_wcl, ntomo=None, zbins=None, ctomo=0, cneff=1, omdeg=None, prefix=None,
                  inputs=[], outputs=[], stdout=None, stderr=None):
    cmd_line = '''export PYTHONPATH={0}/deswrappers:$PYTHONPATH
{0}/bin/doTomoBinning.py --input {1} --ntomo {ntomo} --zbins {zbins} --ctomo {ctomo} --cneff {cneff} --omdeg {omdeg} --prefix {prefix}
...'''
```

```
@App("bash", dfk)
def TreeCorr (conf, outt)
    # Call doTomoBinning
    fu_dtb, data_dtb = doTomoBinning(d_wlpipe, # Directory containing apps
                                    "{0}/wcl/doTomoBinning-{1}.wcl".format(d_wlpipe, dataset), # WCL file
                                    ntomo=ntbomo, zbins=zbins, omdeg=omdeg, prefix=fileprfx, # Various args
                                    outputs=["%s_neff.txt" % fileprfx] + [{"0}_gg_z{1}.fits".format(fileprfx,i) for i in range(1,ntbomo+1)
                                    stdout="tomobinning.out", stderr="tomobinning.err") # Logging etc.

    # Use file_name
    file_name2 = ""

    cmd_line = '''
nbins={nbins}
... % (file_name

    # Call TreeCorr on combinations
    # ... Calculate 2pcf for Xipm
    # .. Xipm cross correlations
    gg_outputs = []
    for comb in combinations(range(1, nbtomo+1), 2):
        print("Launching TreeCorr GG on %s%s" % comb)
        filename=fileprfx+"_gg%s%s.out" % comb
        fu_tc, data_tc = TreeCorr("%s/cfgs/defTreeCorr.yaml" % d_wlpipe,
                                  cmd_options = 'ra_col="RA" dec_col="DEC" ra_units="degrees" dec_units="degrees" g1_col="S_1" g2_col="S_2"
                                  inputs=[data_dtb[comb[0]], data_dtb[comb[1]]], # Pass in the input data futures from TomoBinning
                                  min_sep=thetamin, max_sep=thetamax, nbins=ntheta, bin_slop=bin_slop, # Args
                                  outputs=[filename], # Specify output files
                                  stdout='gg1_tcorr.%s%s.out' % comb, stderr='gg1_tcorr.%s%s.err' % comb)
        gg_outputs.extend(data_tc)
```

Demo

[\(http://try-parsl.parsl-project.org\)](http://try-parsl.parsl-project.org)

Parsl provides 4 important benefits:

Intuitive programming model in Python

Integrates with the Python ecosystem

Makes parallelism more transparent

Parallel dataflow programming

Makes computing location more transparent

Runs your script on multiple distributed sites and diverse computing resources (desktop to petascale) with transparent data movement

Enables provenance capture

Tasks have recordable inputs and outputs

Conclusion: parallel workflow scripting is practical, productive, and necessary, at a broad range of scales

- Swift programming model demonstrated feasible and scalable on HPC, HTC, cloud systems
- Parsl takes this highly successful model and brings it to Python
 - No porting of existing scripts to other languages
 - Support for both Python and external app functions
- Already applied to numerous MTC and HPC application domains
 - attractive for data-intensive applications
 - and several hybrid programming models
- Deep integration with growing ecosystem:
 - Globus, Python, Jupyter, workflow library, ...

Workflow through implicitly parallel dataflow is productive for applications and systems at many scales, including on highest-end system

Parsl Resources

- Getting started
 - <http://try-parsl.parsl-project.org>
- Parsl tutorial
 - <https://github.com/Parsl/parsl-tutorial>
- Documentation
 - <https://parsl.readthedocs.io/en/latest/>

Questions?

<http://parsl-project.org>

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



U.S. DEPARTMENT OF
ENERGY



THE UNIVERSITY OF
CHICAGO



ILLINOIS
UNIVERSITY OF ILLINOIS AT URBANA CHAMPAIGN