parsl

Parsl: Pervasive Parallel Programing in Python

Daniel S. Katz (d.katz@ieee.org, @danielskatz)

Yadu Babuji, Anna Woodard, Zhuozhao Li, Ben Clifford, Rohan Kumar, Lukasz Lacinski, Ryan Chard, Justin M. Wozniak, Ian Foster, Michael Wilde, Kyle Chard

http://parsl-project.org

THE UNIVERSITY OF CHICAGO

Argonne NATIONAL LABORATORY

ILLINOIS NCSA

Supporting composition and parallelism in Python

Software is increasingly *assembled* rather than written

- High-level language (e.g., Python) to integrate and wrap components from many sources

Parallel and distributed computing is no longer a niche area

- Increasing data sizes combined with plateauing sequential processing power
- Parallel hardware (e.g., accelerators) and distributed computing systems

Parsl allows for the natural expression of parallelism in such a way that programs can express opportunities for parallelism that can then be realized, at execution time, using different execution models on different parallel platforms

Parsl: Interactive parallel programming in Python

Apps define opportunities for parallelism

- 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!
```

Try parsl via binder at bottom left of http://parsl-project.org

Expressing a many task workflow in Parsl

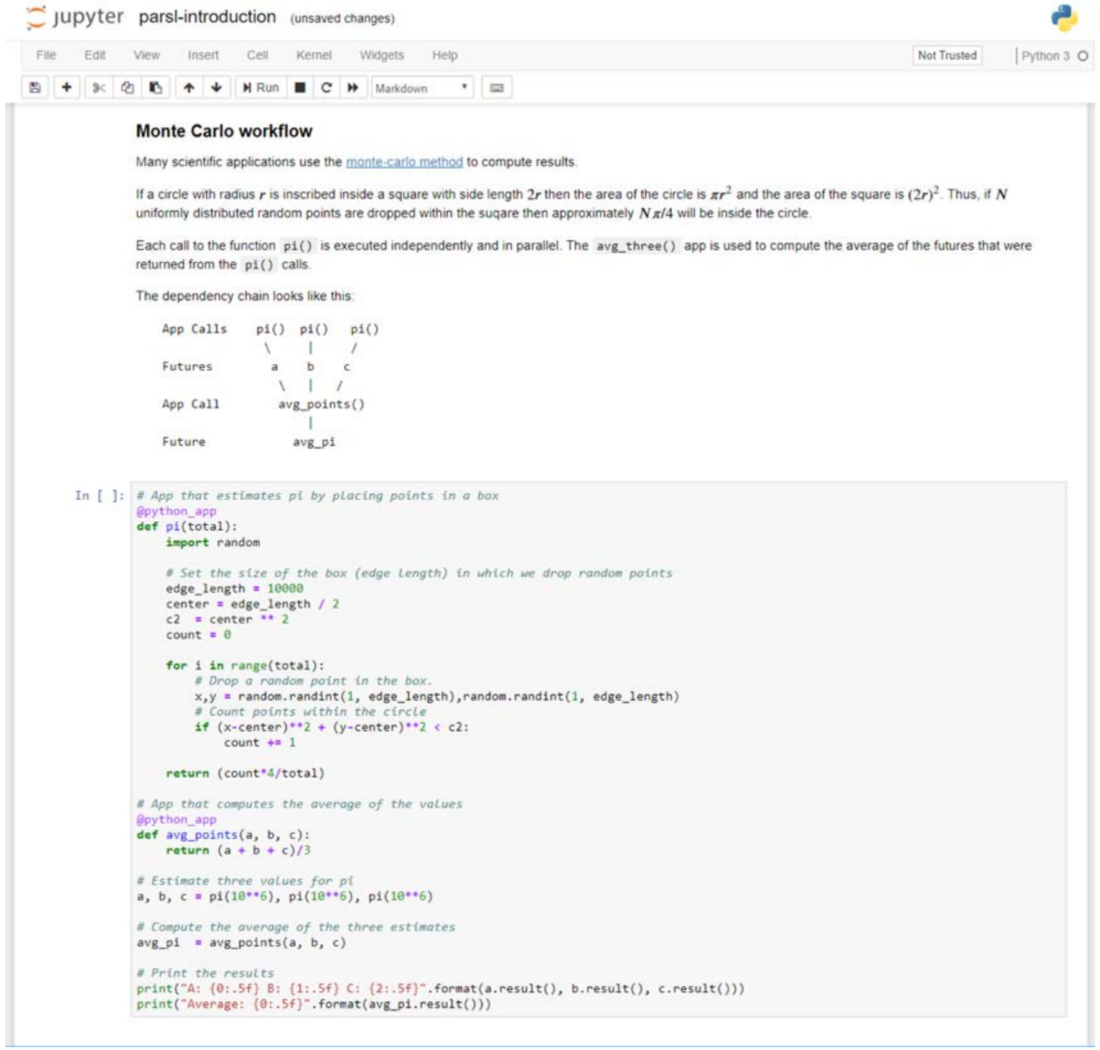
1) Wrap a protein docking code as a Parsl App:

```
@bash_app
def dock(p, c):
   return 'dock.sh {0} {1}'.format(p, c)
```

2) Execute a protein docking workflow by calling Apps:

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

scatter_plot = analyze(structure)
```

Decomposing dynamic parallel execution into a task-dependency graph



jupyter parsl-introduction (unsaved changes)

File Edit View Insert Cell Kernel Widgets Help | Not Trusted | Python 3

Run | Markdown

Monte Carlo workflow

Many scientific applications use the monte-carlo method to compute results.

If a circle with radius r is inscribed inside a square with side length $2r$ then the area of the circle is πr^2 and the area of the square is $(2r)^2$. Thus, if N uniformly distributed random points are dropped within the suqare then approximately $N\pi/4$ will be inside the circle.

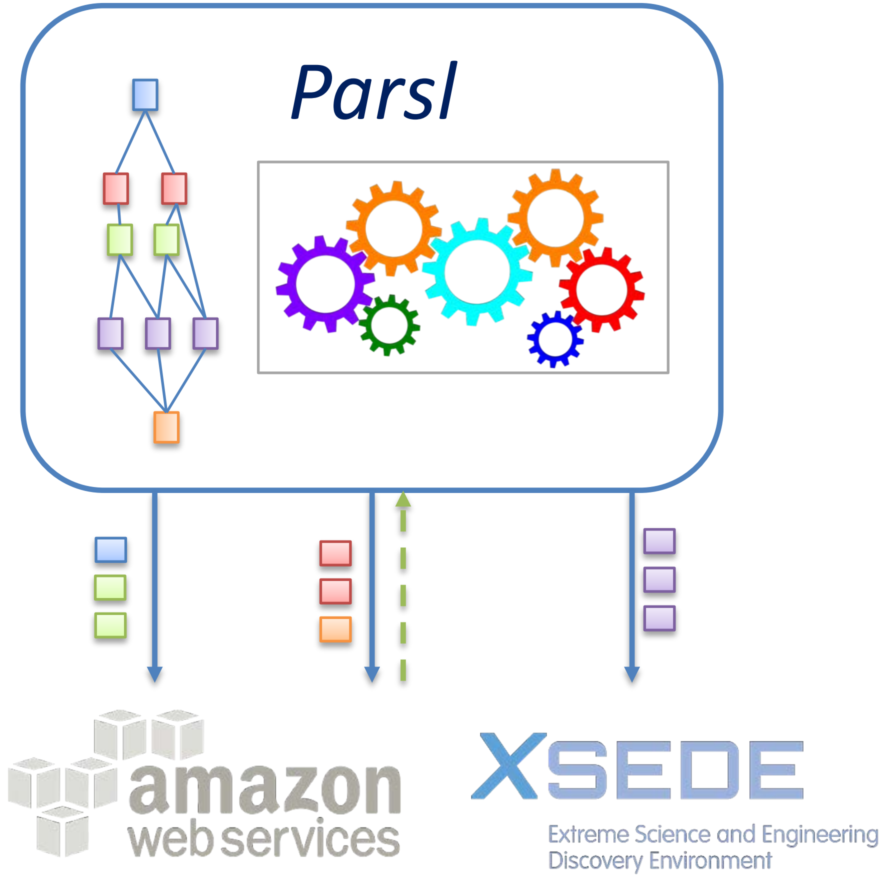
Each call to the function `pi()` is executed independently and in parallel. The `avg_three()` app is used to compute the average of the futures that were returned from the `pi()` calls.

The dependency chain looks like this:

```
App Calls    pi()  pi()   pi()
               \    |    /
Futures         a    b    c
                 \   |   /
App Call        avg_points()
                     |
Future             avg_pi
```

In [ ]:

```python
# App that estimates pi by placing points in a box
@python_app
def pi(total):
    import random

    # Set the size of the box (edge length) in which we drop random points
    edge_length = 10000
    center = edge_length / 2
    c2  = center ** 2
    count = 0

    for i in range(total):
        # Drop a random point in the box.
        x,y = random.randint(1, edge_length),random.randint(1, edge_length)
        # Count points within the circle
        if (x-center)**2 + (y-center)**2 < c2:
            count += 1

    return (count*4/total)

# App that computes the average of the values
@python_app
def avg_points(a, b, c):
    return (a + b + c)/3

# Estimate three values for pi
a, b, c = pi(10**6), pi(10**6), pi(10**6)

# Compute the average of the three estimates
avg_pi  = avg_points(a, b, c)

# Print the results
print("A: {0:.5f} B: {1:.5f} C: {2:.5f}".format(a.result(), b.result(), c.result()))
print("Average: {0:.5f}".format(avg_pi.result()))
```



Parsl scripts are execution provider independent

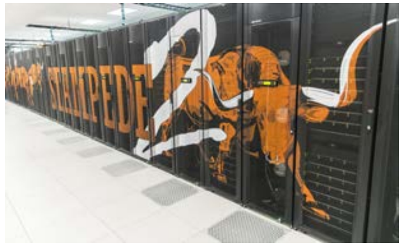
The same script can be run locally, on grids, clouds, or supercomputers

– Works directly with the scheduler (no HTC-like setup)

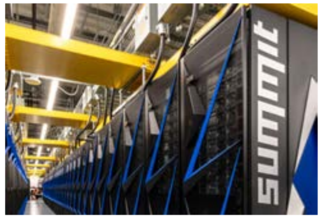
Containers for per-app execution or repeated invocation of the same app

Growing support for various execution providers and resources:

– Local, Cloud (AWS, Azure, private), Slurm, Torque, Condor, Cobalt



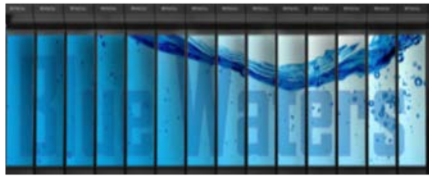
amazon web services



Open Science Grid

Google Cloud

XSEDE



Separation of code and execution

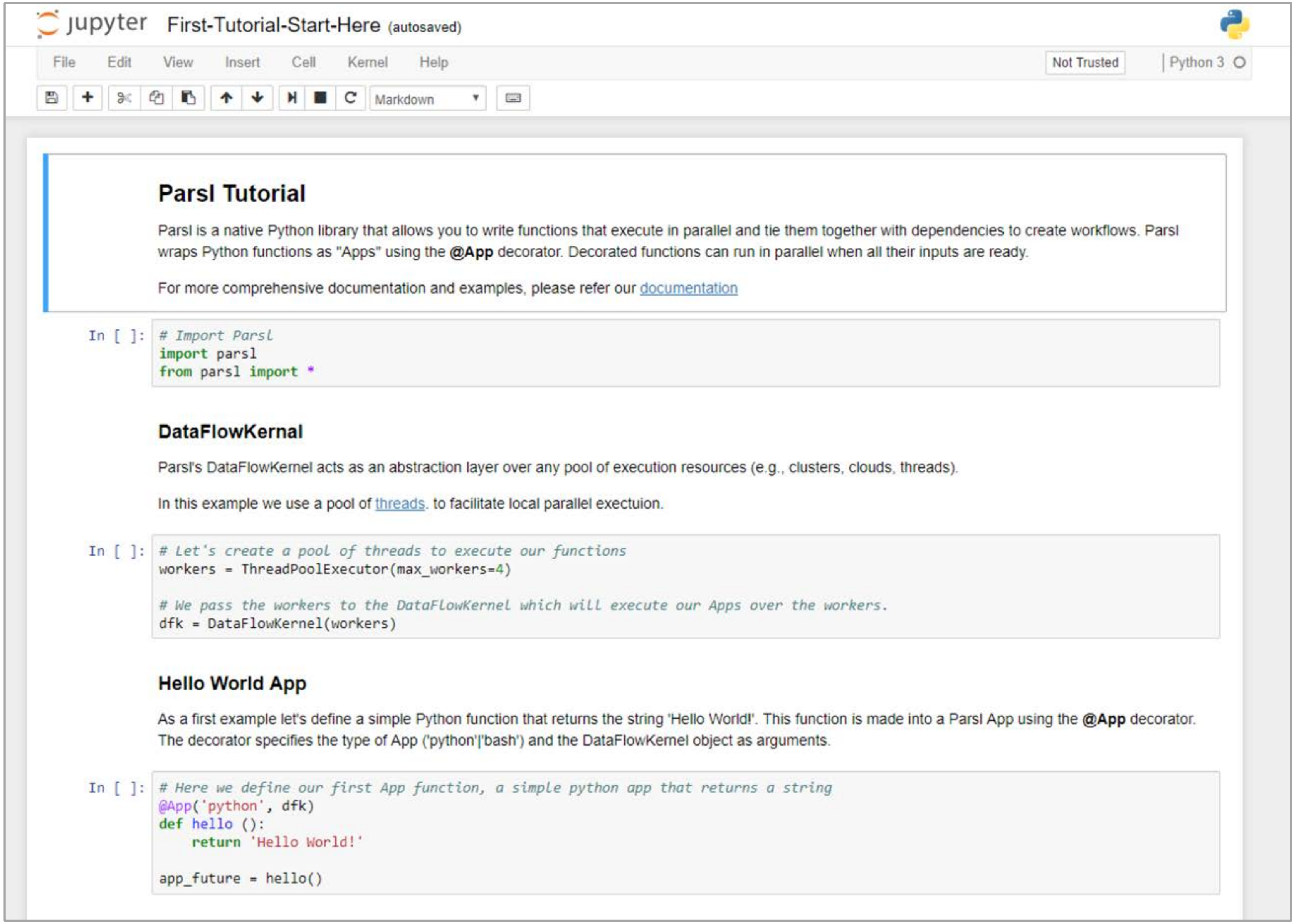
```
from libsubmit.channels import SSHChannel
from libsubmit.providers import SlurmProvider

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)
```

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

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

Pilot jobs on a cluster

Local threads

Interactive supercomputing in Jupyter notebooks



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()
```

Authentication and authorization

Authn/z is hard…

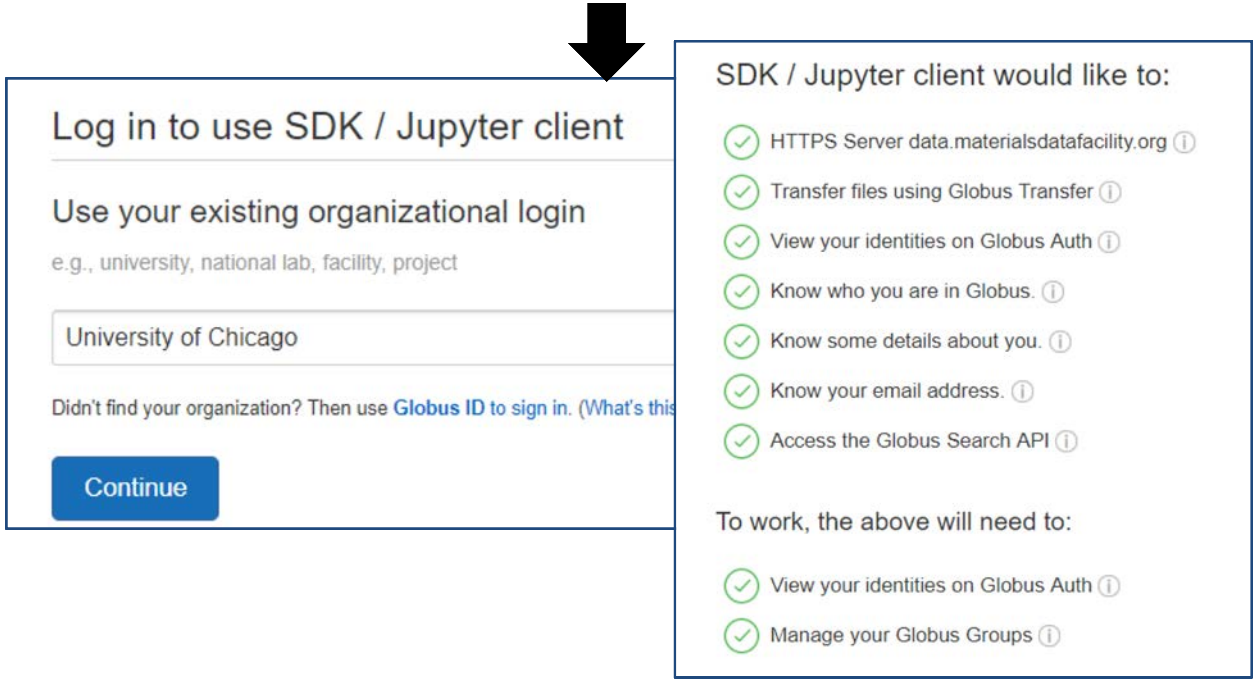
- 2FA, X509, GSISSH, etc.

Integration with Globus Auth to support native app integration for accessing Globus (and other) services

Using scoped access tokens, refresh tokens, delegation support

```
@python_app
def sort_strings(inputs=[], outputs=[]):
    with open(inputs[0], 'r') as u:
        strs = u.readlines()
        strs.sort()
        with open(outputs[0].filepath, 'w') as s:
            for e in strs:
                s.write(e)

unsorted_globus_file = File('globus://03d7d06a-cb6b-11e8-8c6a-0a1d4c5c824a/unsorted.txt')
sorted_globus_file = File('globus://d59900ef-6d04-11e5-ba46-22000b92c6ec/sorted.txt')

f = sort_strings(inputs=[unsorted_globus_file], outputs=[sorted_globus_file])
print (f.result())
parsl.clear()
```

```
Please visit the following URL to provide authorization:
https://auth.globus.org/v2/oauth2/authorize?client_id=8b8060fd-610e-4a74-885e-1051c71ad473&redirect_uri=https%3A%2F%2Fauth.globus.org%2Fv2%2Fweb%2Fauth-code&scope=openid+urn%3Aglobus%3Aauth%3Ascope%3Atransfer.api.globus.org%3Aall&state=_default&response_type=code&code_challenge=wouAVozLGvpaUcEa1_EwwsKsVUFxIthAeurvtSTJwYk&code_challenge_method=S256&access_type=offline

Enter the auth code:
```



Parsl provides transparent (wide area) data management

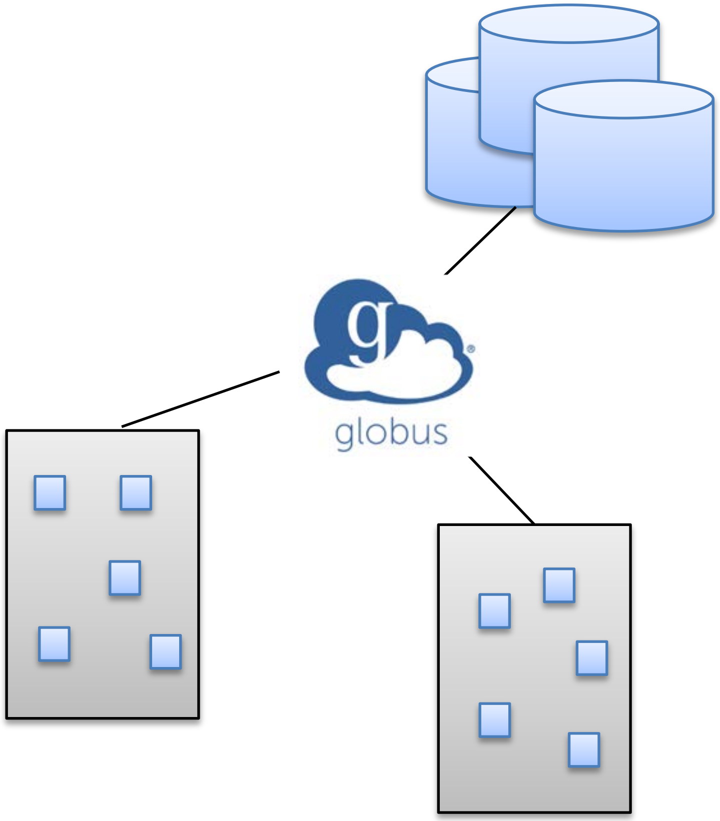
Implicit data movement to/from repositories, laptops, supercomputers

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

Globus for third-party, high performance and reliable data transfer

- Support for site-specific DTNs

HTTP/FTP direct data staging



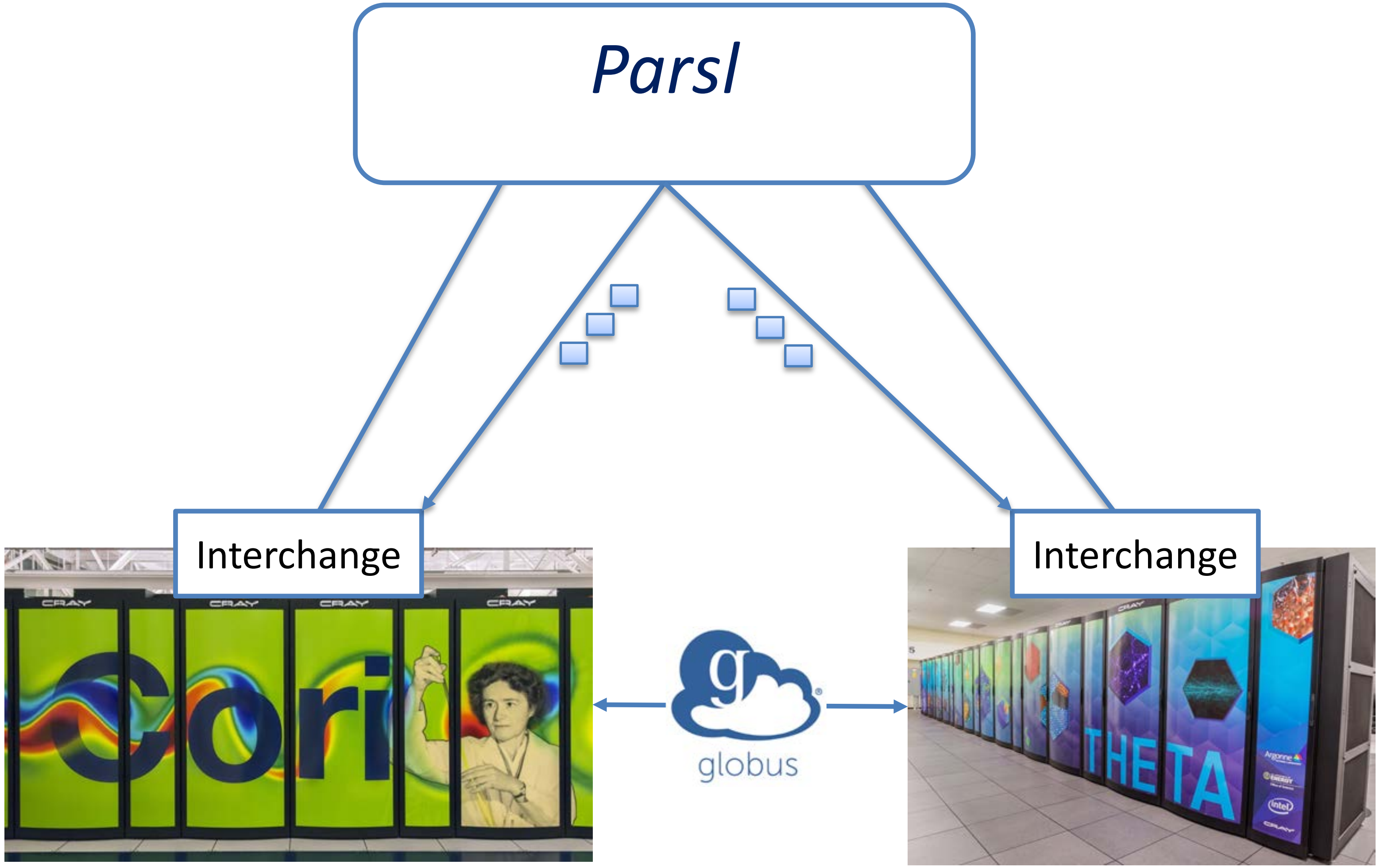
www.globus.org

DOE Distributed Computing & Data Ecosystem (DCDE)

- A DOE group is identifying best practices and research challenges to create and operate a DOE/SC wide federated Distributed Computing & Data Ecosystem (DCDE)
  - Future Lab Computing Working Group (FLC-WG)
  - Initially working towards a pilot
- Using OAuth, working with Globus
  - Test deployment at BNL
- Parsl is part of this effort, via initial work in linking ORNL and BNL
  - We've added support for an OAuthSSHChannel
  - Now being tested on test deployment

Multi-site execution

1. Loading Parsl configuration triggers:
   a) Creation of SSH channels
   b) Deployment of an interchange process onto login nodes
   c) Submission of pilot jobs that will connect to the interchange
2. Parsl submits tasks directly to interchange
3. Parsl uses Globus to stage data



Multi-site execution

```python
from parsl.config import Config

from parsl.providers import SlurmProvider, CobaltProvider
from parsl.executors import HighThroughputExecutor
from parsl.channels import SSHInteractiveLoginChannel
from parsl.data_provider.scheme import GlobusScheme

config = Config(
    executors=[
        HighThroughputExecutor(
            label='theta',
            max_workers=4,
            address='try.parsl-project.org',
            interchange_address='thetalogin6',
            provider=CobaltProvider(
                channel=SSHInteractiveLoginChannel(
                    hostname="thetalogin6.alcf.anl.gov",
                    username="yadunand",            # MUST SET PER USER
                    script_dir="/home/yadunand/parsl_scripts"
                ),
                queue="debug-flat-quad",
                init_blocks=1,
                min_blocks=1,
                worker_init='source /home/yadunand/setup_parsl_0.7.2.sh',
                account='CSC249ADCD01',             # MUST SET PER USER
                cmd_timeout=120
            ),
            working_dir='/home/yadunand',
            storage_access=[GlobusScheme(
                endpoint_uuid='08925f04-569f-11e7-bef8-22000b9a448b',
                endpoint_path='/',
                local_path='/')],
        ),
        HighThroughputExecutor(
            label="cori",
            worker_debug=False,
            address='try.parsl-project.org',
            interchange_address='cori03-224.nersc.gov',
            provider=SlurmProvider(
                partition='debug',  # Replace with partition name
                # channel=SSHChannel(
                channel=SSHInteractiveLoginChannel(
                    hostname='cori03-224.nersc.gov',
                    username='yadunand',            # MUST SET PER USER
                    script_dir='/global/homes/y/yadunand/parsl_scripts',
                ),
                [illegible]
                scheduler_options="#SBATCH --constraint=knl,quad,cache",
                # scheduler_options="#SBATCH --constraint=haswell",
                worker_init='source ~/setup_parsl_0.7.2.sh',
            ),
            working_dir='/global/homes/y/yadunand',
            storage_access=[GlobusScheme(
                endpoint_uuid='9d6d99eb-6d04-11e5-ba46-22000b92c6ec',
                endpoint_path='/',
                local_path='/')],
        )
```

```python
@python_app(executors=['theta'])
def platform_theta(sleep=10, stdout=None):
    import platform
    import time
    time.sleep(sleep)
    return platform.uname()

@python_app(executors=['cori'])
def platform_cori(sleep=10, stdout=None):
    import platform
    import time
    time.sleep(sleep)
    return platform.uname()

@python_app
def double(x):
    return x * 2

theta_p = platform_theta()
cori_p = platform_cori()

[illegible]
print("From theta: ", theta_p.result())
print("From cori: ", cori_p.result())
```

Too much small code

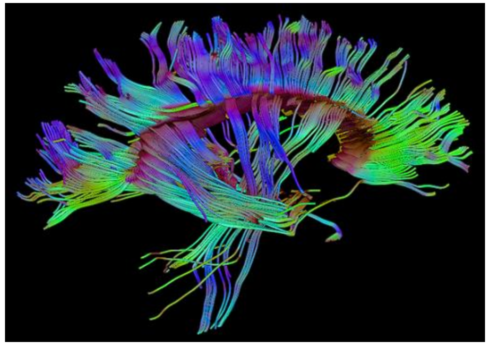
See demo instead

https://bit.ly/2Wsjlep

(code in https://github.com/Parsl/demo_multifacility)

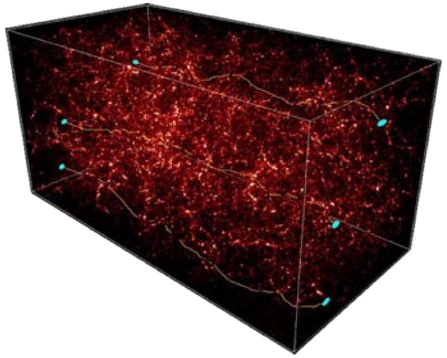
Parallel applications are very different

High-throughput workloads



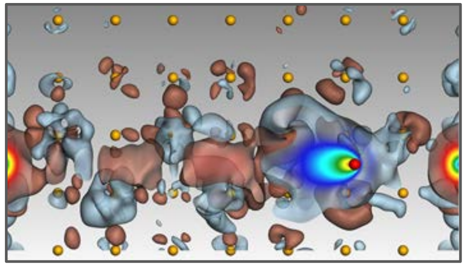
- Protein docking, image processing, materials reconstructions
- **Requirements**: 1000s of tasks, 100s of nodes, reliability, usability, monitoring, elasticity, etc.

Extreme-scale workloads



- Cosmology simulations, imaging the arctic, genomics analysis
- **Requirements**: millions of tasks, 1000s of nodes (100,000s cores), capacity

Interactive and real-time workloads



- Materials science, cosmic ray shower analysis, machine learning inference
- **Requirements**: 10s of nodes, rapid response, pipelining

Parsl's modular executor interface supports these different use cases

High-throughput executor (HTEX)

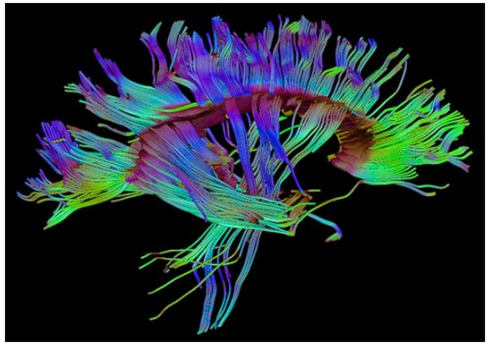
- Designed for ease of use, support for clusters and clouds, fault-tolerance
- <2000 nodes (~60K workers), 1M tasks, task duration/nodes > 0.01 (e.g., with 10 nodes, tasks 100ms)

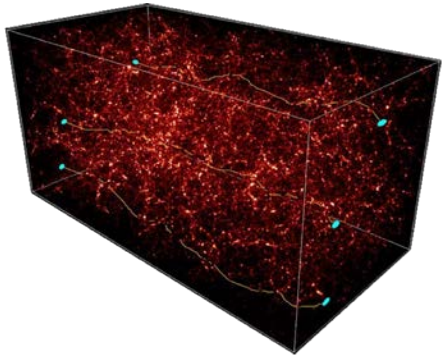
Extreme-scale executor (EXEX)

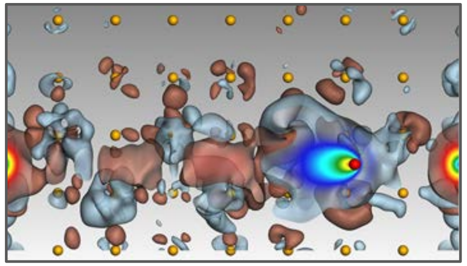
- Distributed MPI job manages execution. Manager rank communicates workload to other worker ranks directly
- >1000 nodes (>30K workers), 1M tasks, >1m task duration

Low-latency executor (LLEX)

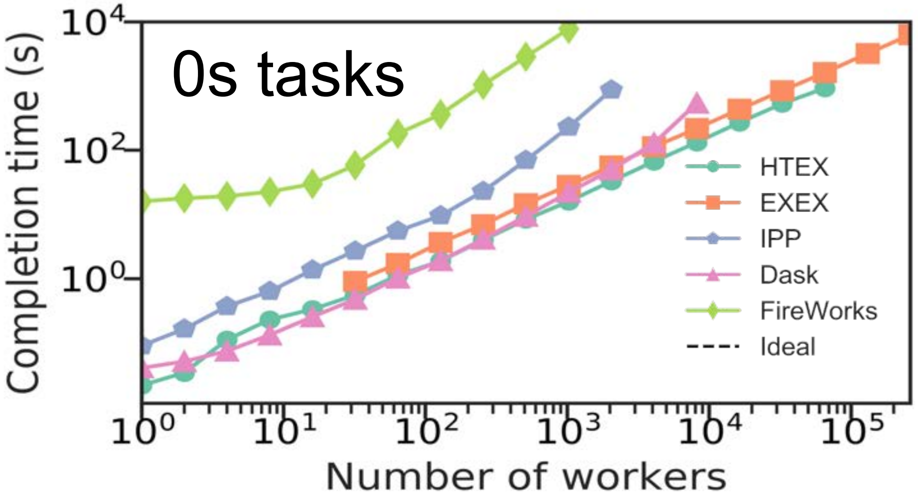
- Barebones executor, assumes small, fixed resource pool, no fault-tolerance, elasticity, etc.
- <10 nodes, <1M tasks







Parsl executors scale to 2M tasks/256K workers

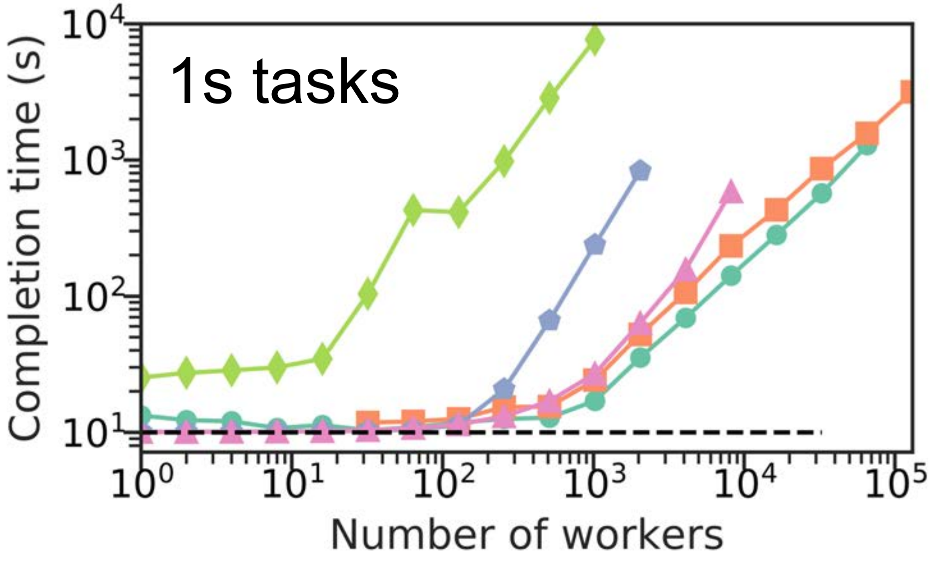


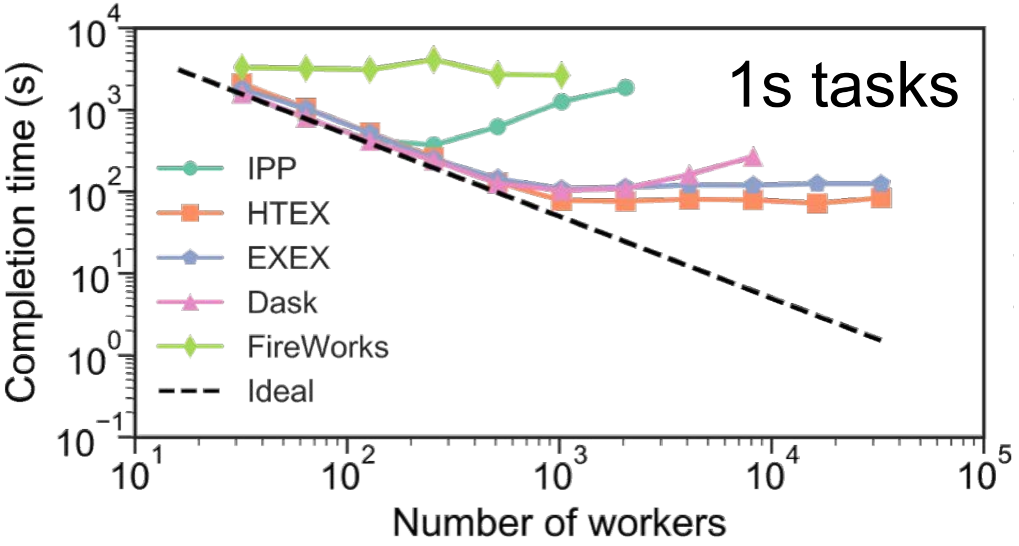
Weak scaling: HTEX & EXEX outperform other approaches up to ~1M tasks

Weak scaling: HTEX & EXEX scale to 2K & 8K nodes, with >1K tasks/s

Strong scaling: HTEX & EXEX outperform other approaches at >256 workers

Framework	Maximum # of workers†	Maximum # of nodes†	Maximum tasks/second‡
Parsl-IPP	2048	64	330
Parsl-HTEX	65 536	2048*	1181
Parsl-EXEX	262 144	8192*	1176
FireWorks	1024	32	4
Dask distributed	4096	128	2617





parsl Weak scaling: 10 tasks per worker

Strong scaling: 50,000 tasks

Other functionality provided by Parsl

Resource abstraction. Block-based model overlaying different providers and resources

Fault tolerance. Support for retries, checkpointing, and memoization

Multi site. Combining executors/providers for execution across different resources

Elasticity. Automated resource expansion/retraction based on workload

Monitoring. Workflow and resource monitoring and visualization

Globus. Delegated authentication and wide area data management

Data management. Automated staging with HTTP, FTP, and Globus

Containers. Sandboxed execution environments for workers and tasks

jupyter

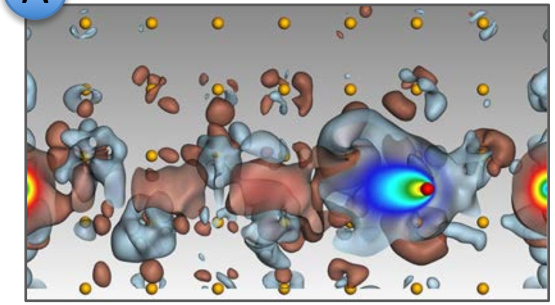
Jupyter integration. Seamless description and management of workflows

Reproducibility. Capture of workflow provenance in the task graph

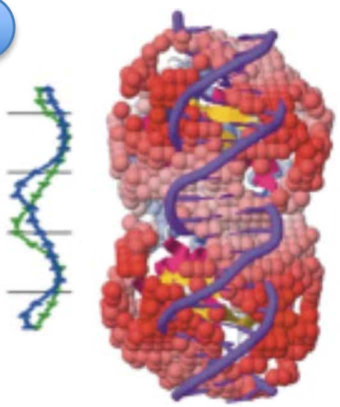
Parsl is being used in a wide range of scientific applications

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

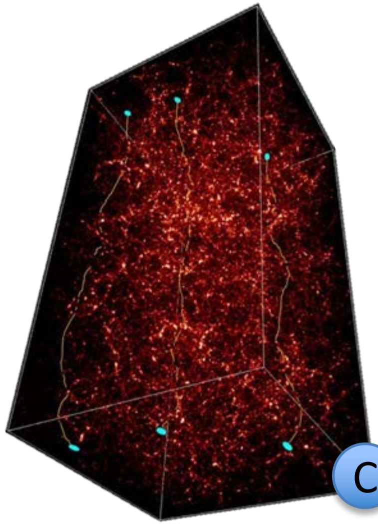
A



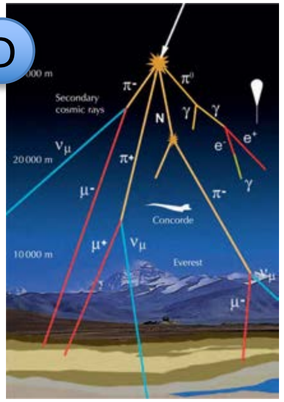
B



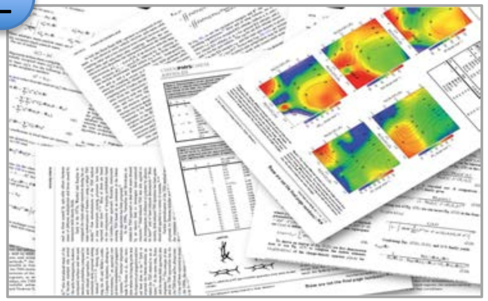
C



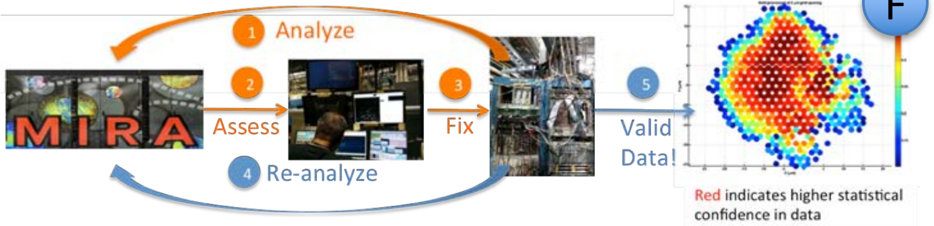
D



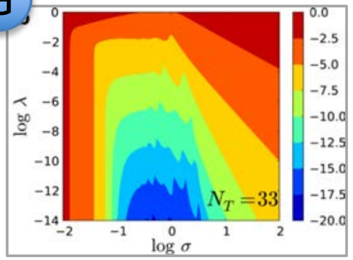
E



F



G



Resource configuration

- Execution environment configured via Config object(s), e.g.

```
import parsl
from parsl.config import Config
from parsl.executors.threads import ThreadPoolExecutor

config = Config(
    executors=[ThreadPoolExecutor()],
    lazy_errors=True
)
parsl.load(config)
```

- Based on: where tasks execute; which executor; where main Parsl program executes, which provider, which launcher
- Examples in Parsl documentation (→), but

Note

Please note that all configuration examples below require customization for your account, allocation, Python environment, etc.

⊟ Configuration

- How-to Configure
- Comet (SDSC)
- Cori (NERSC)
- Stampede2 (TACC)
- Theta (ALCF)
- Cooley (ALCF)
- Swan (Cray)
- CC-IN2P3
- Midway (RCC, UChicago)
- Open Science Grid
- Amazon Web Services
- Ad-Hoc Clusters
- Further help

Community Challenges

- Describing HPC and other remote systems
  - Do we all have to do this separately? With all users maintaining knowledge of their systems
  - Can we build something like Globus endpoints that are maintained by system folks?
  - Do batchspawner and wrapspawner help?
- Describing applications
  - We wrap apps that we run on HPC systems for Parsl
  - Others “wrap” them differently (CWL, Pegasus, etc.)
  - Can we come up with a common method for this?

Parsl provides simple, safe, scalable, and flexible parallelism in Python

Simple: Python with minimal new constructs (integrated with the growing SciPy ecosystem and other scientific services), works in Jupyter

Safe: deterministic parallel programs through immutable input/output objects, dependency task graph, etc.

Scalable: efficient execution from laptops to (multiple of) the largest supercomputers

Flexible: programs composed from existing components and then applied to different resources/workloads

Developer friendly: open source, on GitHub, Apache 2 license, collaborators welcome

Parsl 0.8.0a out now (142 s) – moving towards 1.0!

Questions?

http://parsl-project.org
(binder link to try parsl at bottom left)

Parsl paper (HPDC-28, June 26 2019)
https://doi.org/10.1145/3307681.3325400
preprint: https://arxiv.org/abs/1905.02158

amazon web services
U.S. DEPARTMENT OF ENERGY
NSF
THE UNIVERSITY OF CHICAGO
Argonne NATIONAL LABORATORY
ILLINOIS NCSA