ExaWorks

https://exaworks.org

Tutorial: Developing Robust and Scalable Next Generation Workflows Applications and Systems

ISC-HPC 2022

Lawrence Livermore National Laboratory

Argonne NATIONAL LABORATORY

Brookhaven National Laboratory

OAK RIDGE National Laboratory

U.S. DEPARTMENT OF ENERGY | Office of Science

Parsl

https://parsl-project.org/

Parsl: a parallel programming library for Python

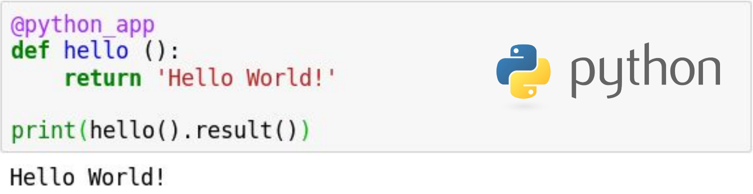
Apps define opportunities for parallelism
Python apps call Python functions
Bash apps call external applications

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

Apps run concurrently respecting dataflow dependencies. Natural parallel programming!

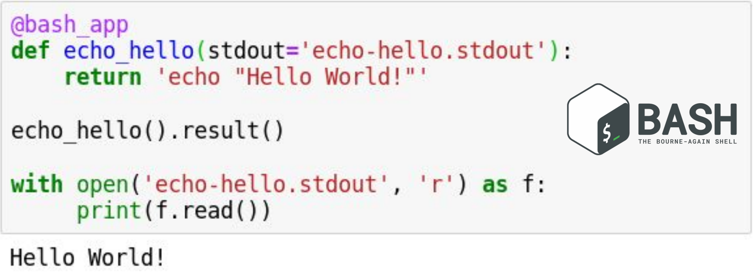
Parsl scripts are independent of where they run. Write once run anywhere!

```
pip install parsl
```



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

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

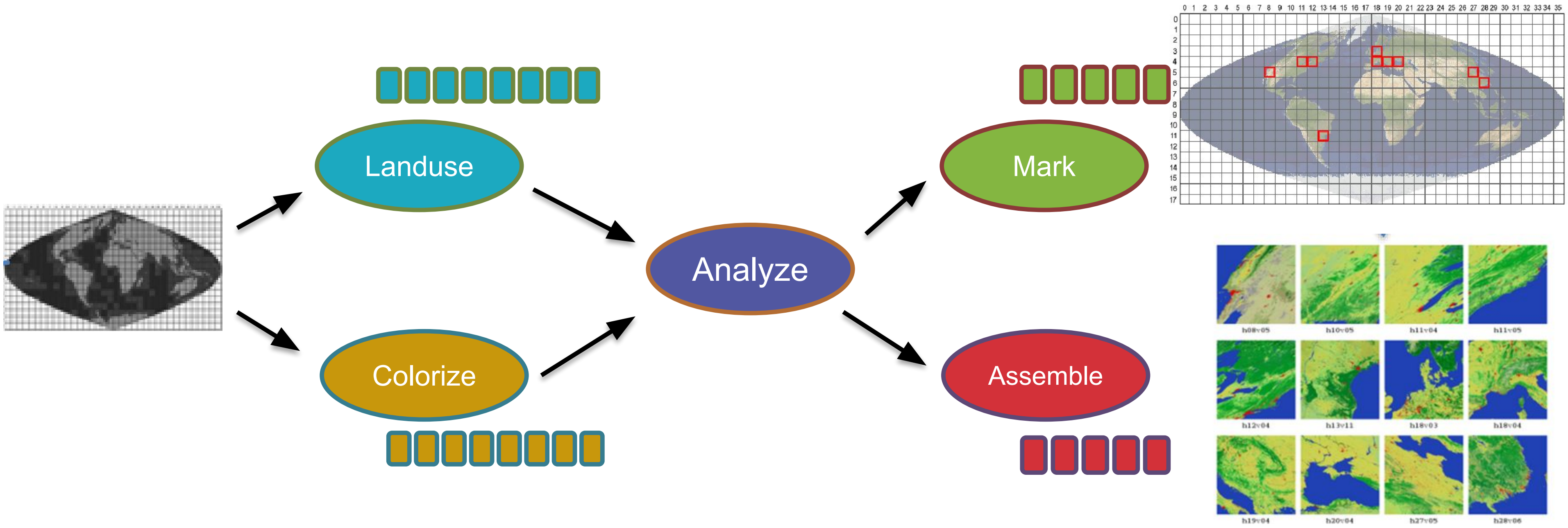
```
Hello World!
```



```
@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!
```

Data-driven example: parallel geospatial analysis



Land-use Image processing pipeline for the MODIS remote sensor

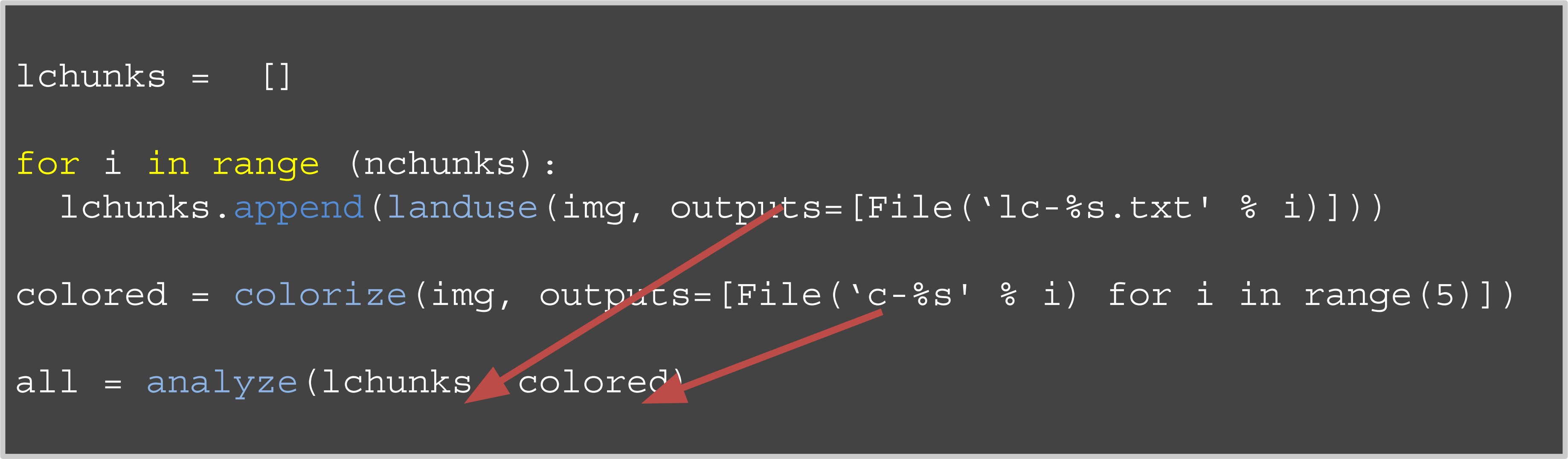
Expressing parallelism using Parsl

1) Wrap the science applications as Parsl Apps:

```
@bash_app
def landuse(img, outputs=[]):
  return ‘./landuse_sim.sh {} {}’.format(img, outputs[0])

@python_app
def colorize(img, outputs=[]):
  return color_package(img, len(outputs))

@python_app
def analyze(land_chunks, color_chunks):
  return combine(land_chunks, color_chunks)
```

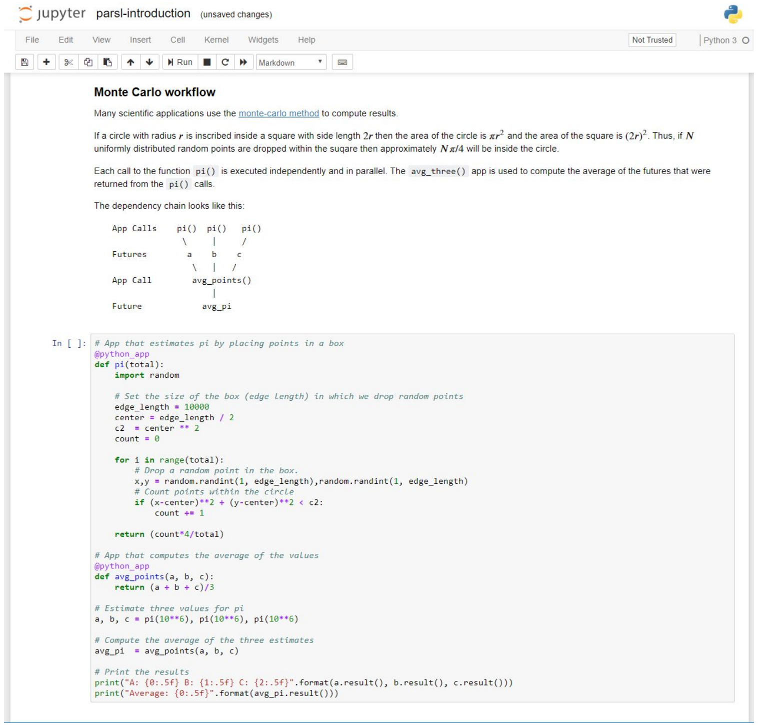
Expressing a many task workflow in Parsl

2) Execute the parallel workflow by calling Apps:



```
lchunks =   []

for i in range (nchunks):
  lchunks.append(landuse(img, outputs=[File(‘lc-%s.txt' % i)]))

colored = colorize(img, outputs=[File(‘c-%s' % i) for i in range(5)])

all = analyze(lchunks, colored)
```

Decomposing dynamic parallel execution into a task-dependency graph



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.

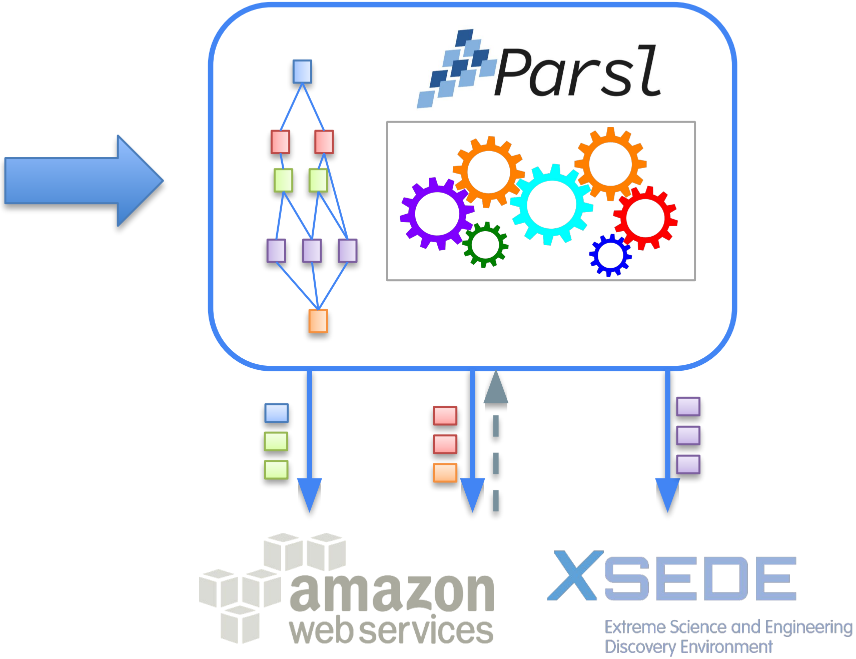
Each call to the function `pi()` is executed independently and in parallel. The `avg_three()` app is used to compute the average of the futures that were returned from the `pi()` calls.

The dependency chain looks like this:

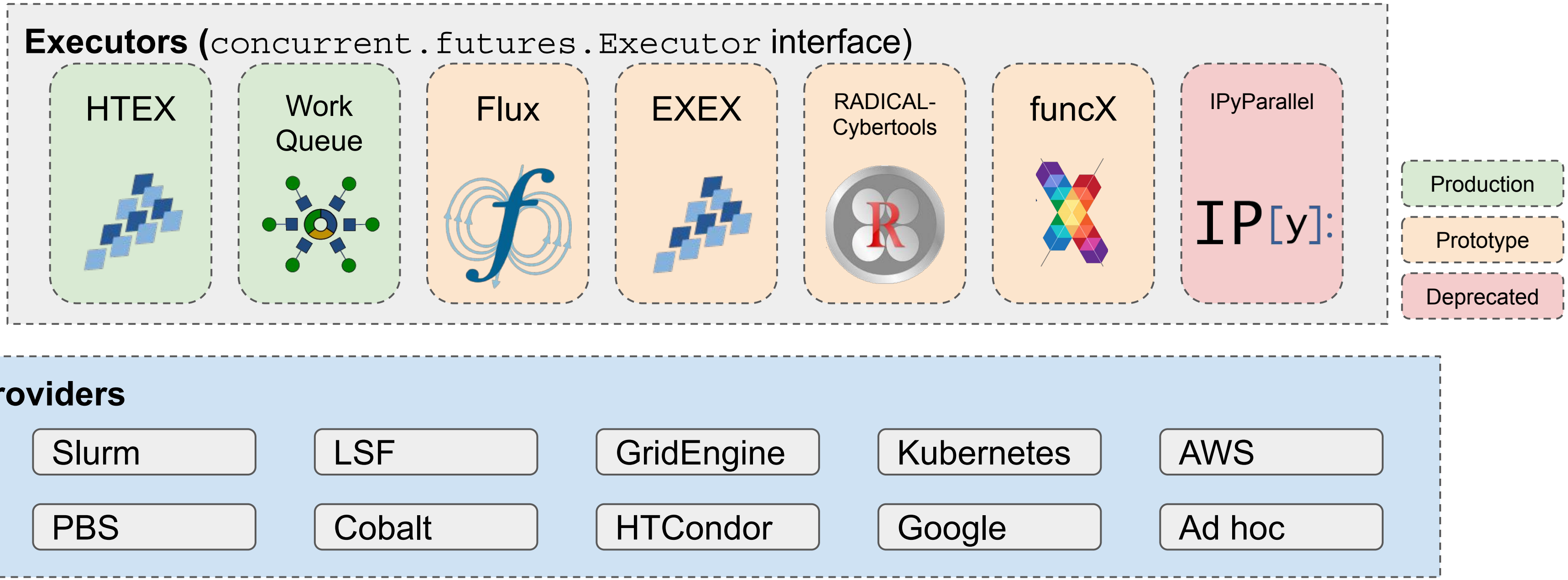
```
App Calls    pi()  pi()   pi()
               \     |     /
Futures         a    b    c
                 \   |   /
App Call        avg_points()
                     |
Future             avg_pi
```

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 programs can be executed in different ways on different systems



Executors (`concurrent.futures.Executor` interface)

- HTEX
- Work Queue
- Flux
- EXEX
- RADICAL-Cybertools
- funcX
- IPyParallel

- Production
- Prototype
- Deprecated

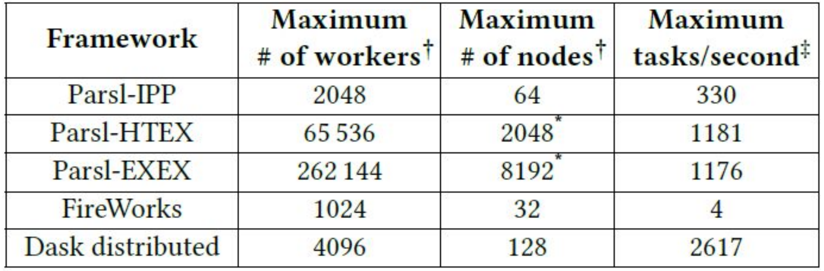
Providers

- Slurm
- LSF
- GridEngine
- Kubernetes
- AWS
- PBS
- Cobalt
- HTCondor
- Google
- Ad hoc

Parsl executors scale to 2M tasks/256K workers

HTEX and EXEX outperform other Python-based approaches

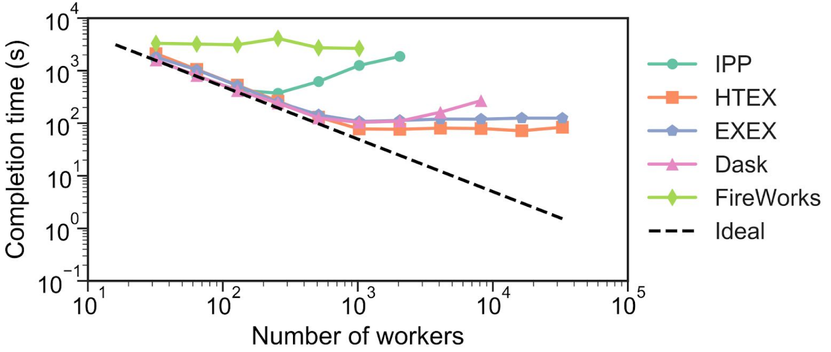
Parsl scales to more than 250K workers (8K nodes) and ~2M tasks



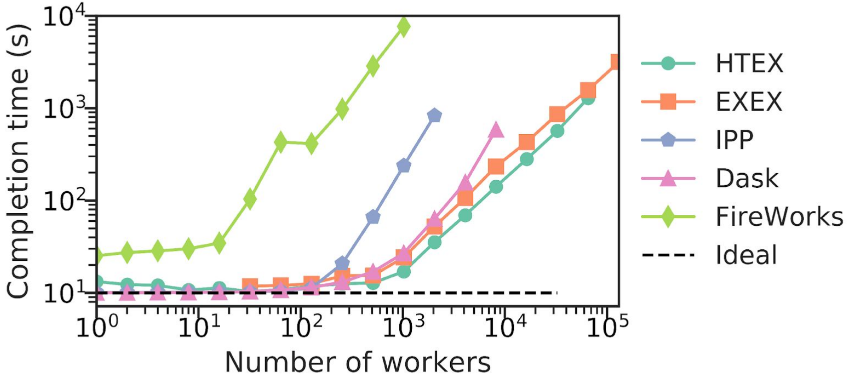
Framework	Maximum # of workers†	Maximum # of nodes†	Maximum tasks/second‡
Parsl-IPP	2048	64	330
Parsl-HTEX	65 536	2048*	1181
Parsl-EXEX	262 144	8192*	1176
FireWorks	1024	32	4
Dask distributed	4096	128	2617

Babuji et.al. "Parsl: Pervasive Parallel Programming in Python." ACM International Symposium on High-Performance Parallel and Distributed Computing (HPDC). 2019.

Strong scaling (50K 1s tasks)



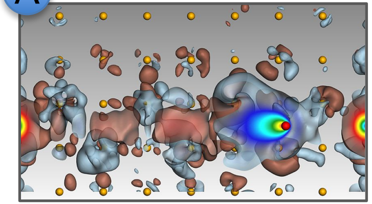
Weak scaling (10 1s tasks per worker)



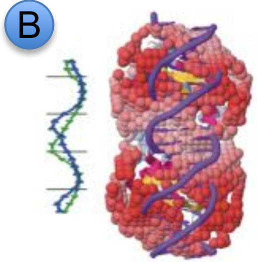
Parsl is being used in a wide range of scientific applications

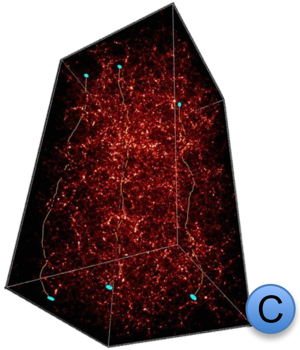
- A Machine learning to predict stopping power in materials
- B Protein and biomolecule structure and interaction
- C LSST simulation and weak lensing using sky surveys
- D Cosmic ray showers in QuarkNet
- E Information extraction to classify image types in papers
- F Materials science at the Advanced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

A



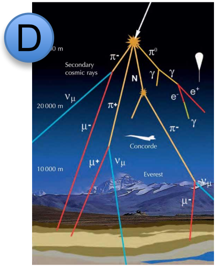
B



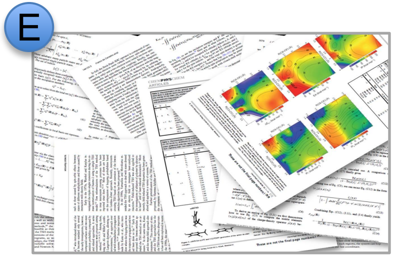


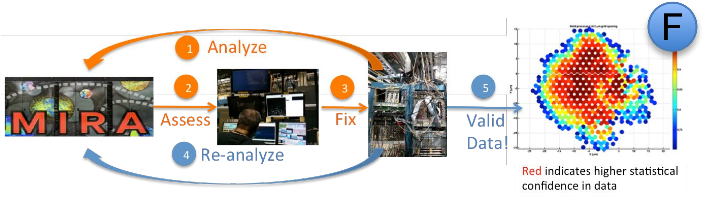
C

D



E

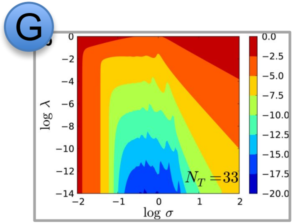




F

Red indicates higher statistical confidence in data

G



More examples:
`https://parsl-project.org/parslfest`

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 integration

Seamless description and management of workflows

Reproducibility

Capture of workflow provenance in the task graph

Exercises

`https://github.com/ExaWorks/Tutorial`