Parsl: Pervasive Parallel Programming in Python

Kyle Chard (chard@uchicago.edu)

Yadu Babuji, Anna Woodard, Ben Clifford, Zhuozhao Li, Mike Wilde, Dan Katz, Ian Foster

http://parsl-project.org

THE UNIVERSITY OF CHICAGO

Argonne NATIONAL LABORATORY

ILLINOIS UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

Composition and parallelism

(Scientific) software is increasingly *assembled* rather than written

- High-level language to integrate and wrap components from many sources

Parallel and distributed computing is ubiquitous

- Increasing data sizes combined with plateauing sequential processing power

Python (and the SciPy ecosystem) is the de facto standard language (for science)

- Libraries, tools, Jupyter, etc.

Parsl allows for the natural expression of parallelism in Python:

- Programs can express opportunities for parallelism
- Realized, at execution time, using different execution models on different parallel platforms

Fourth Generation Parallel Dataflow Scripting

2001 ***Virtual Data Language*** *original declarative effort*

2006 ***Swift/K*** *http://swift-lang.org*

Very fast, highly portable, pervasively parallel dataflow
Orchestrates apps passing files

2009 ***Swift/T*** *http://swift-lang.org/Swift-T*

Ultra scalable, distributed interpretation, MPI-based
Adds in-memory functions and datasets

2017 ***Parsl parallel programming library*** *http://parsl-project.org*

All of the above, in Python

Parsl: parallel programming in Python

Apps define opportunities for parallelism

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

```
pip install parsl
```

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

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

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

```
Hello World!
```

Apps run concurrently respecting data dependencies. Natural parallel programming!

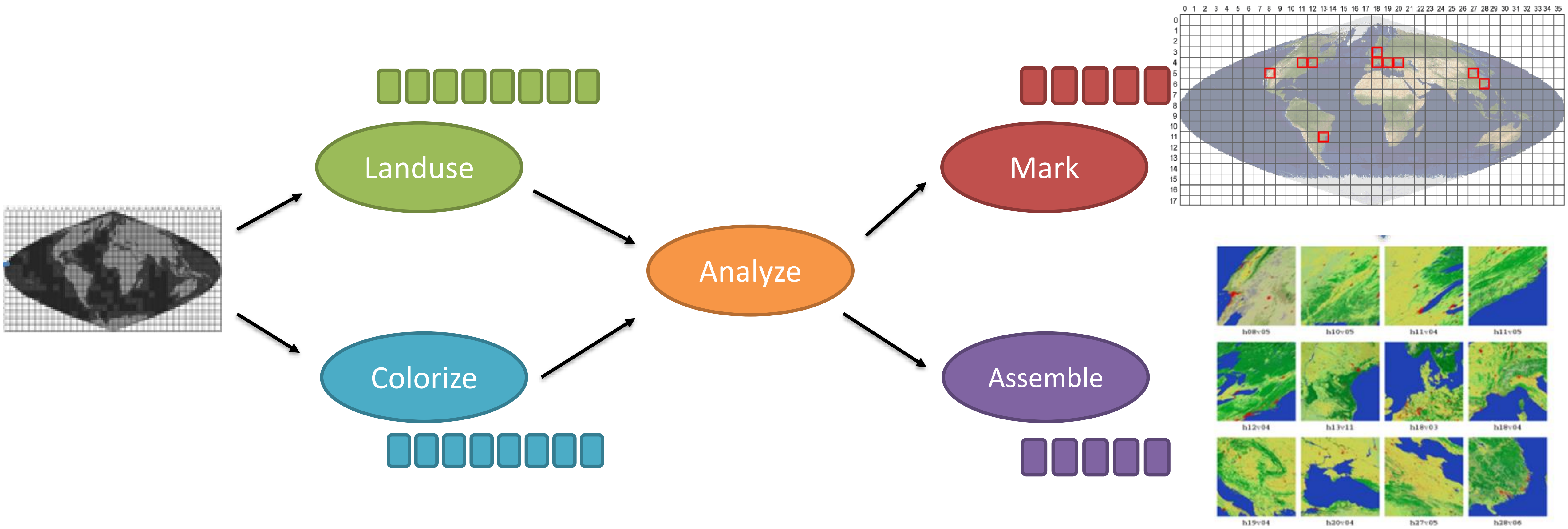
```
@bash_app
def echo_hello(stdout='echo-hello.stdout'):
    return 'echo "Hello World!"'

echo_hello().result()

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

```
Hello World!
```

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

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

Data-driven example: parallel geospatial analysis



Land-use Image processing pipeline for the MODIS remote sensor

Expressing a many task workflow in Parsl

1) Wrap the science applications as Parsl Apps:

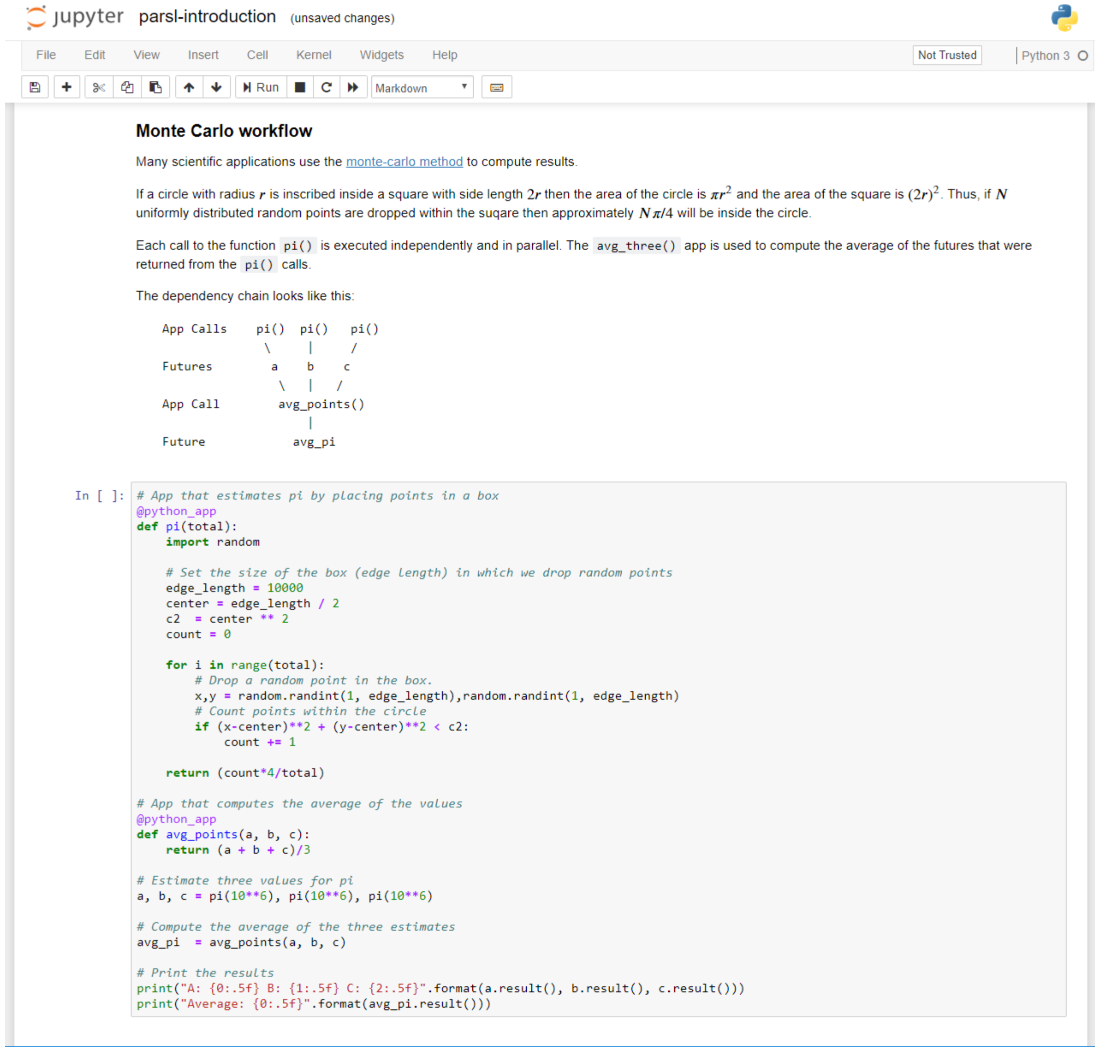
```
@bash_app
def simulate(outputs=[]):
    return './simulation_app.exe {outputs[0]}’

@bash_app
def merge(inputs=[], outputs=[]):
    i = inputs; o = outputs
    return './merge {1} {0}'.format(' '.join(i), o[0])

@python_app
def analyze(inputs=[]):
    return analysis_package(inputs)
```

Expressing a many task workflow in Parsl

2) Execute the parallel workflow by calling Apps:

```
sims = []

for i in range (nsims):
     sims.append(simulate(outputs=['sim-%s.txt' % i]))

all = merge(inputs=[i.outputs[0] for i in sims],
            outputs=['all.txt'])

result = analyze(inputs=[all.outputs[0]])
```

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.

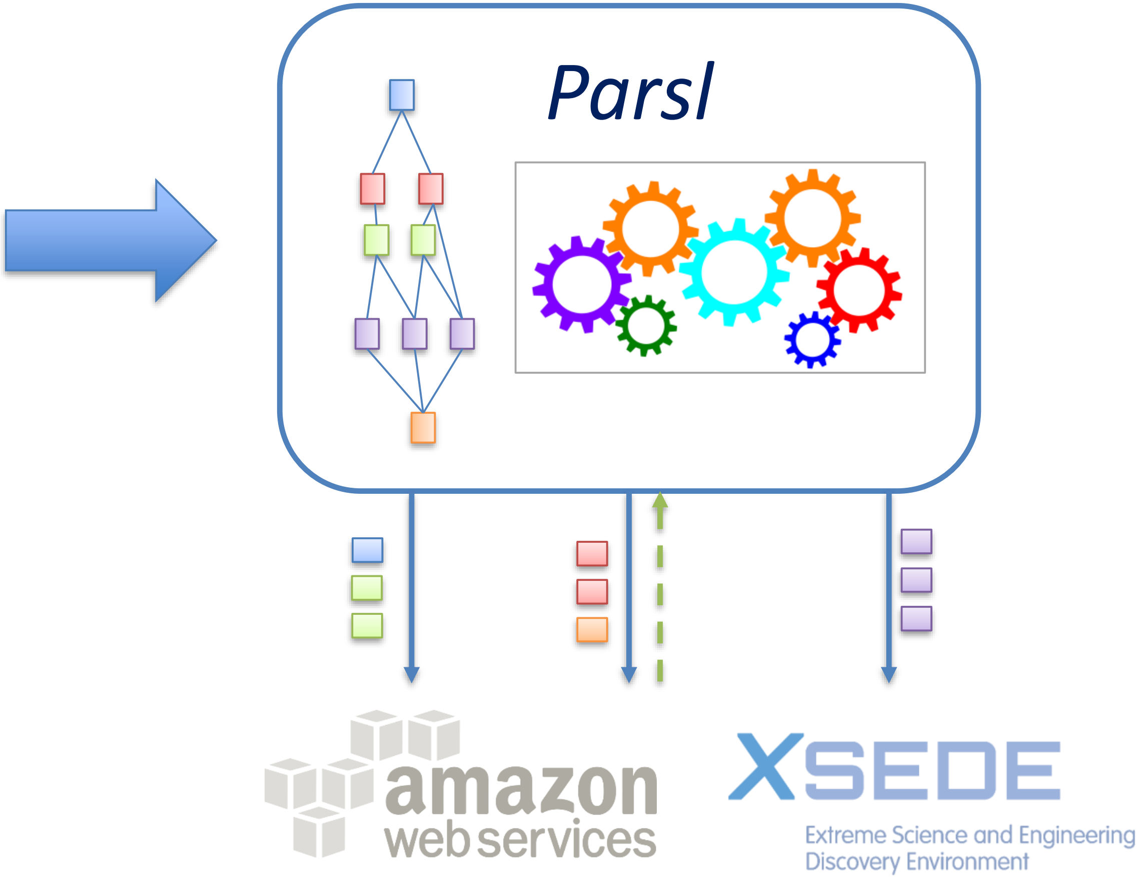
Each call to the function `pi()` is executed independently and in parallel. The `avg_three()` app is used to compute the average of the futures that were returned from the `pi()` calls.

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

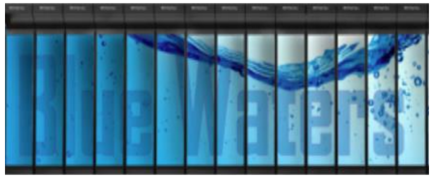
Growing support for various schedulers and cloud vendors

⊟ **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

Open Science Grid

amazon web services



Google Cloud

XSEDE

parsl

Separation of code and execution

sample_configs.py

```python
# ... imports

threads_config = Config(
    executors=[ThreadPoolExecutor()]
)

cori_config = Config(
    executors=[
        HighThroughputExecutor(
            label='Cori_HTEX_multinode',
            provider=SlurmProvider(
                'debug',   # Partition / QOS
                nodes_per_block=2,
                walltime="00:20:00",
                launcher=SrunLauncher()
            ))
    ])
```

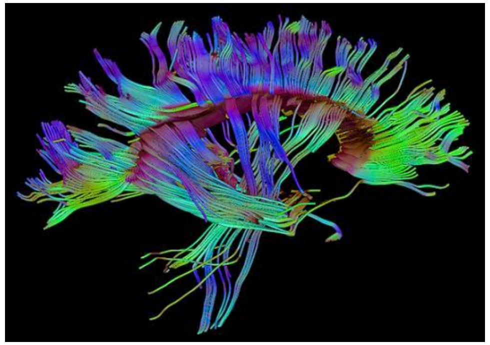
runner.py

```python
import parsl
import os
from sample_configs import threads_config, cori_config

if os.environ.get('PIPELINE_ENV', 'test'):
    parsl.load(threads_config)
else:
    parsl.load(cori_config)

#... rest of the pipeline...
```

Choose execution environment at runtime. Parsl will direct tasks to the configured execution environment(s).

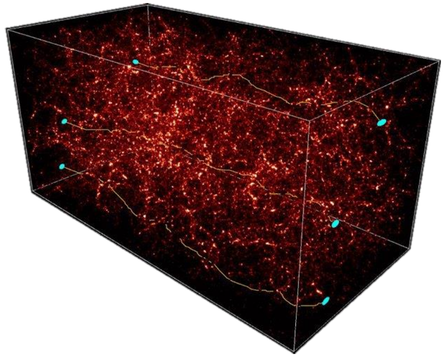
parsl

Parallel applications require different execution models



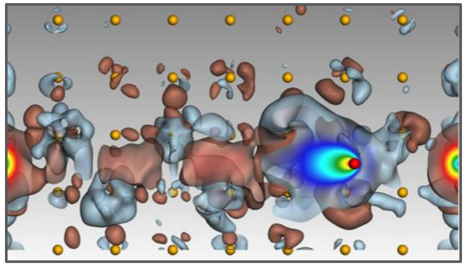
High-throughput workloads

- Protein docking, image processing, materials reconstructions
- **Requirements**: 1000s of tasks, 100s of nodes, days of execution, 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), days of execution, capacity



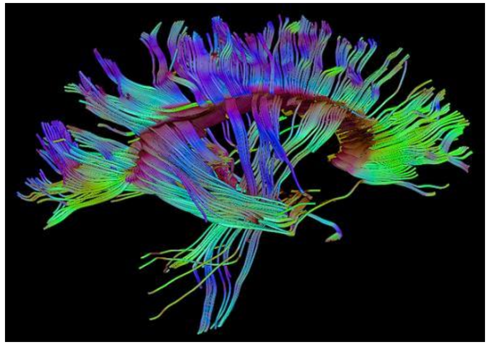
Interactive and real-time workloads

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

Parsl implements an extensible executor interface

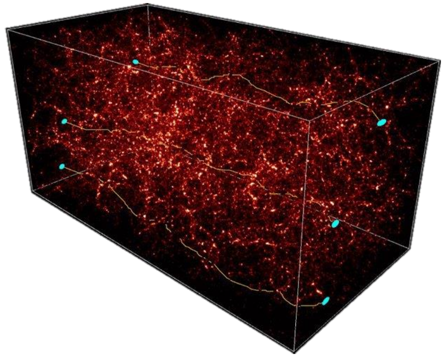
High-throughput executor (HTEX)

- Pilot job-based model with multi-threaded manager deployed on workers
- Designed for ease of use, fault-tolerance, etc.
- <2000 nodes (~60K workers), Ms tasks, task duration/nodes > 0.01



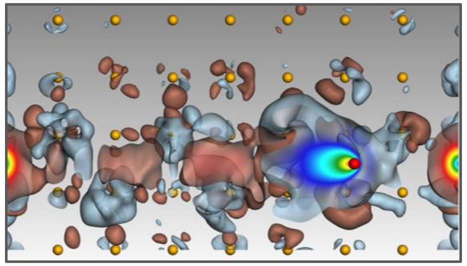
Extreme-scale executor (EXEX)*

- Distributed MPI job manages execution. Manager rank communicates workload to other worker ranks directly
- Designed for extreme scale execution on supercomputers
- >1000 nodes (>30K workers), Ms tasks, >1m task duration



Low-latency Executor (LLEX)*

- Direct socket communication to workers, fixed resource pool, limited features
- 10s nodes, <1M tasks, <1m tasks

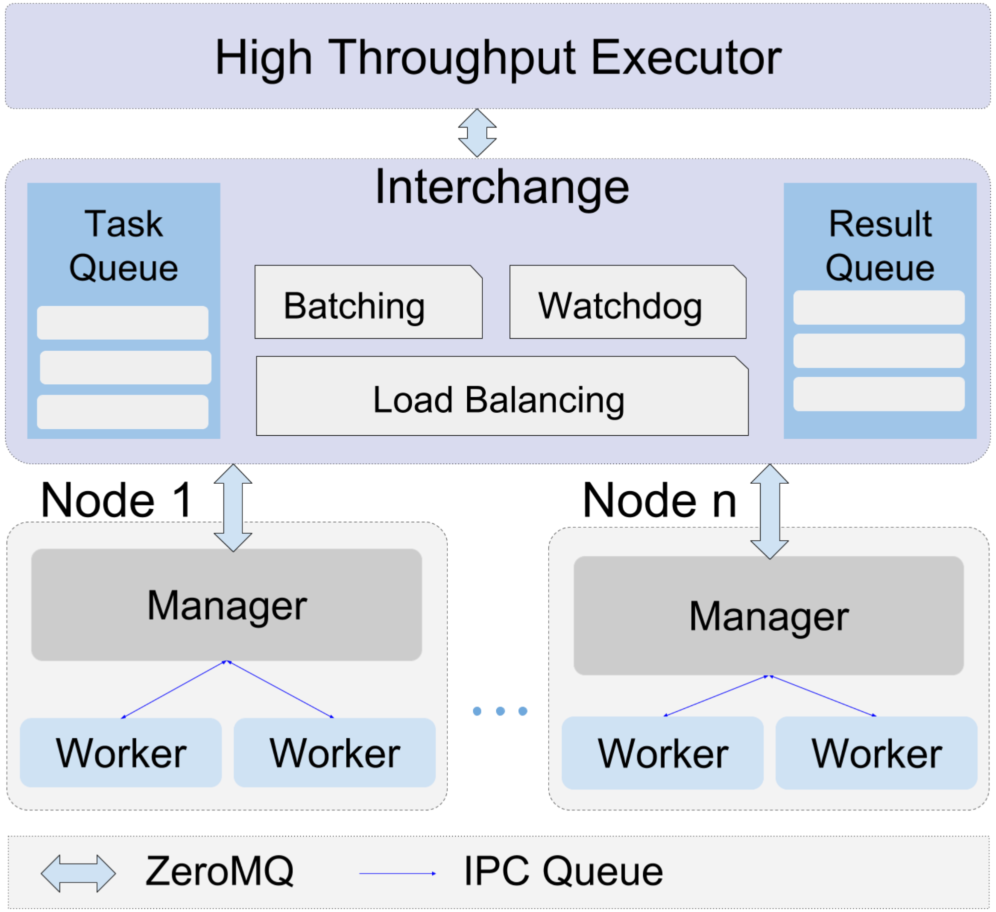


parsl

Dissecting the High-throughput Executor

Pilot job-based execution with a multi-threaded manager deployed on each worker

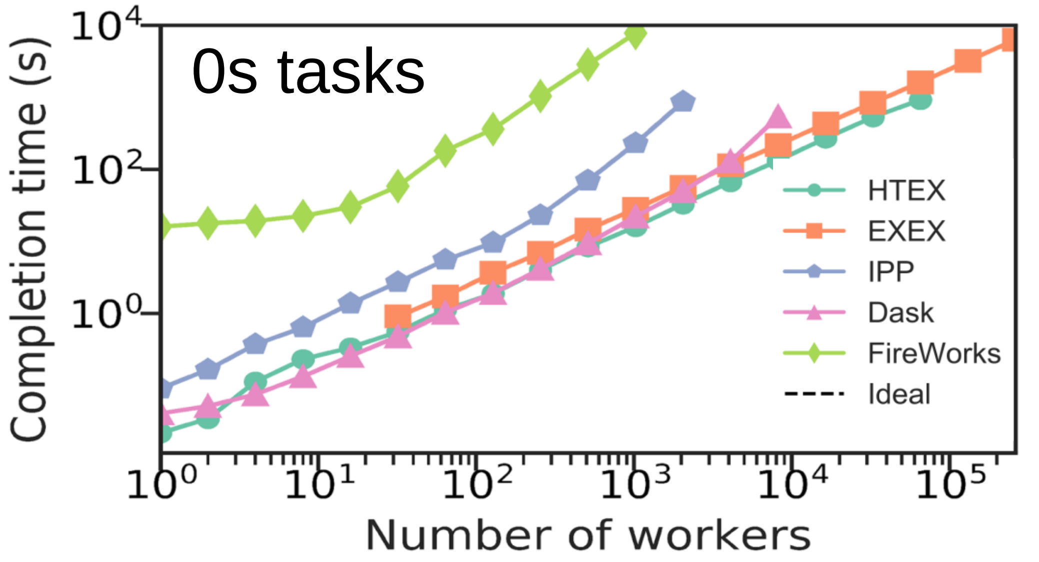
Interchange queues and processes messages to/from manager via two queues (sockets)



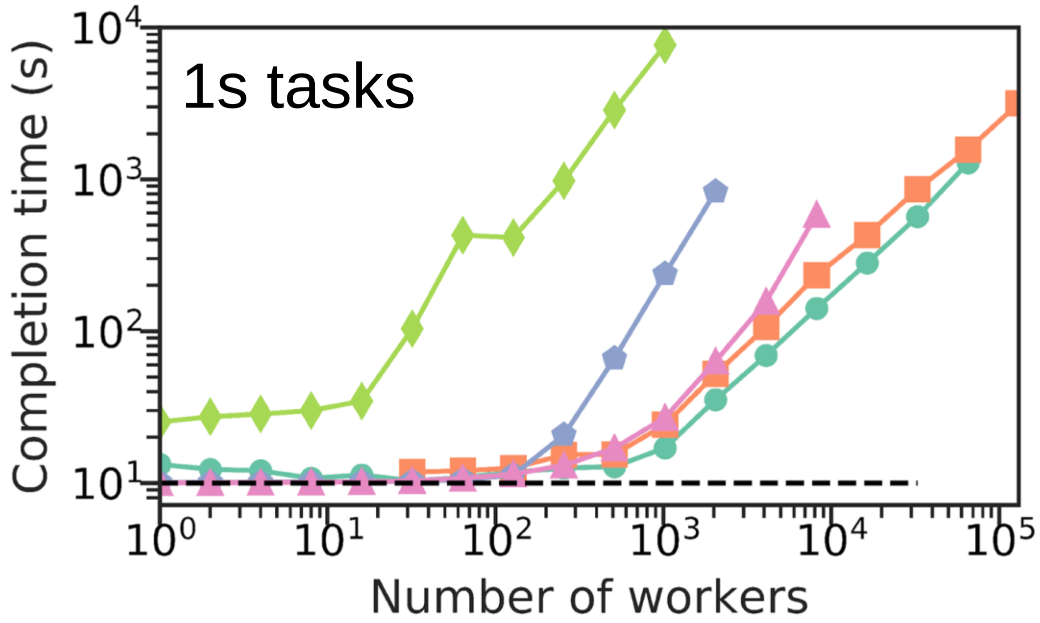
Parsl executors scale to 2M tasks/256K workers

Weak scaling: 10 tasks per worker

- HTEX and EXEX outperform other Python-based approaches and scale beyond ~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.

Monitoring and visualization

Parsl Workflows Documentation

Workflows

Name	Version	Owner	Status	Runtime (s)	Tasks	Actions
test_udp_simple.py	2019-02-20 22:16:43.570094	zhuozhao	Completed	25.218577	5 0	
test_fan_in_out.py	2019-02-20 22:20:24.918435	zhuozhao	Completed	151.207859	12 0	
test_monitoring.py	2019-02-20 22:23:16.632888	zhuozhao	Completed	121.393285	20 0	
test_fan_in_out.py	2019-02-20 22:27:05.407903	zhuozhao	Completed	151.513495	12 0	

test_fan_in_out.py

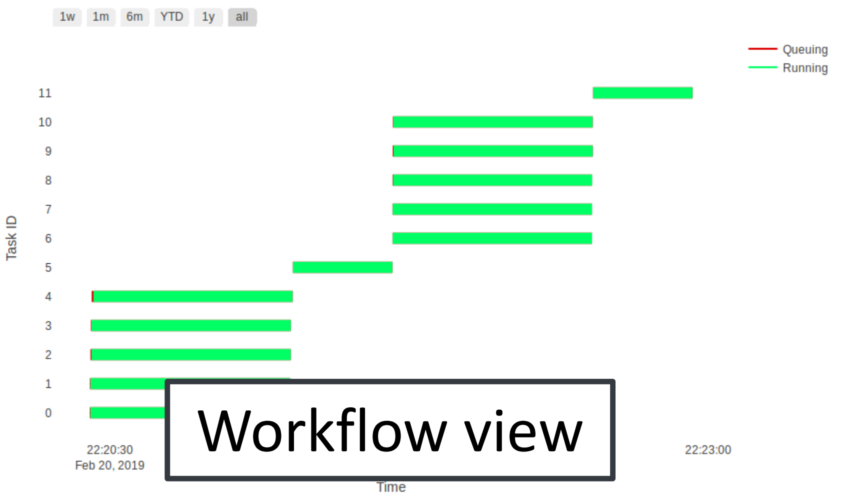
Workflow Summary

- **Started:** 2019-02-20 22:20:24.918435
- **Completed:** 2019-02-20 22:22:56.126294
- **Completion time:** 151.207859 s
- **Owner:** zhuozhao
- **host:** midway2-login2.rcc.local
- **rundir:** /home/zhuozhao/parsl/parsl/tests/manual_tests/runinfo/001
- **tasks_failed_count:** 0
- **tasks_completed_count:** 12

View workflow resource usage

App Summary

Name	Count
add_inc	2
inc	10



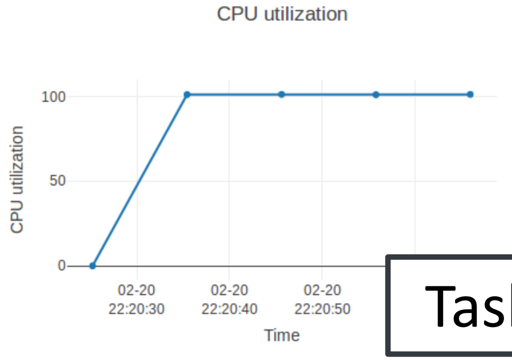
Workflow view

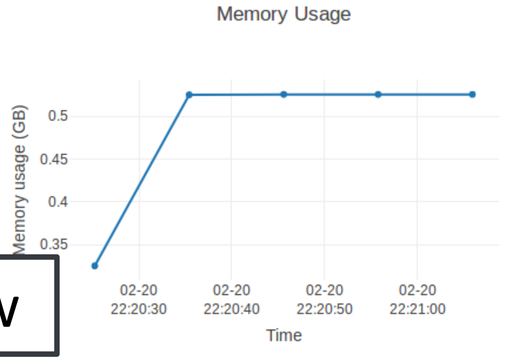
inc (1)

- **Workflow name:** test_fan_in_out.py
- **Started:** 2019-02-20 22:20:24.918435
- **Completed:** 2019-02-20 22:22:56.126294
- **Completion time:** 151.207859 s
- **Owner:** zhuozhao
- **task_func_name:** inc
- **task_id:** 1
- **task_time_submitted:** 2019-02-20 22:20:25.112977
- **task_time_returned:** 2019-02-20 22:21:15.349654
- **task_inputs:** None
- **task_outputs:** None
- **task_stdin:** None
- **task_stdout:** None

Task State

Time	State
2019-02-20 22:20:25.128896	launched
2019-02-20 22:20:25.236034	running
2019-02-20 22:21:15.349689	done





Task view

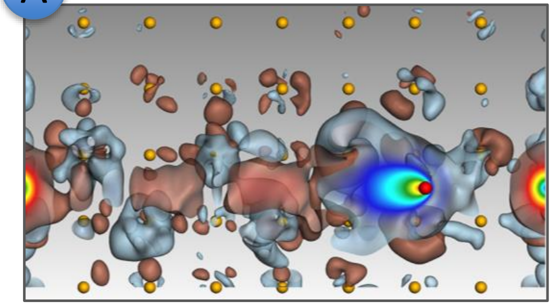
parsl

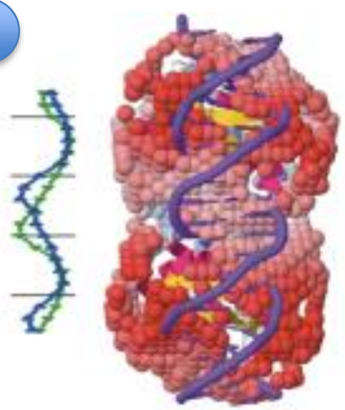
Other functionality provided by Parsl

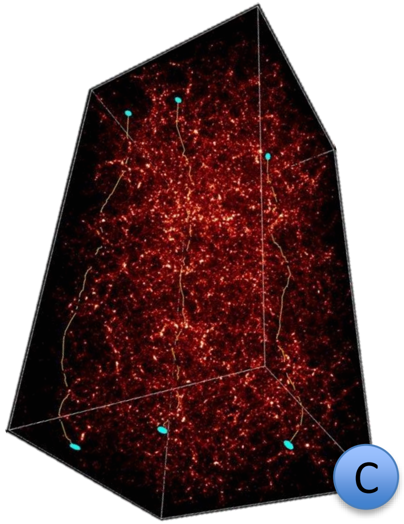
- Resource abstraction. Block-based model overlaying different providers and resources
- Fault tolerance. Support for retries, checkpointing, and memoization
- Multi site. Combining executors/providers for execution across different resources
- Elasticity. Automated resource expansion/retraction based 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 workflow provenance in the task graph

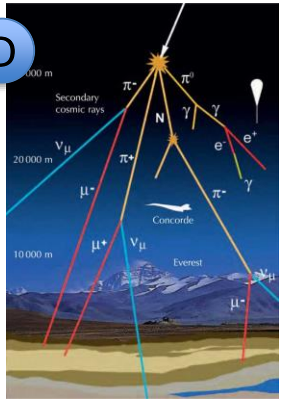
Parsl is being used in a wide range of scientific applications

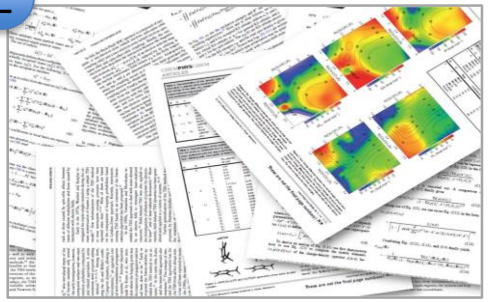
- **A** Machine learning to predict stopping power in materials
- **B** Protein and biomolecule structure and interaction
- **C** Weak lensing using sky surveys
- **D** Cosmic ray showers as part of QuarkNet
- **E** Information extraction to classify image types in papers
- **F** Materials science at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

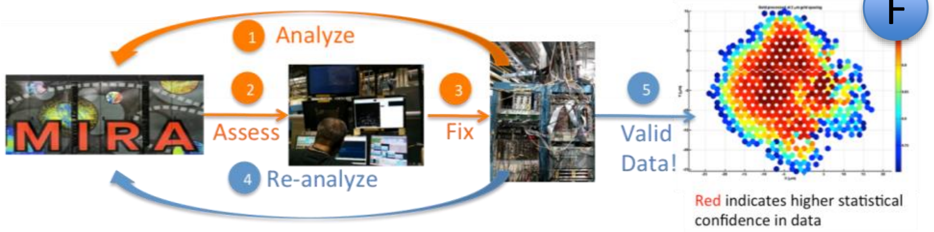
A


B


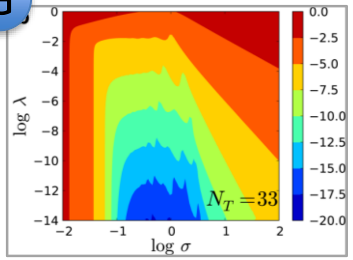

C

D


E


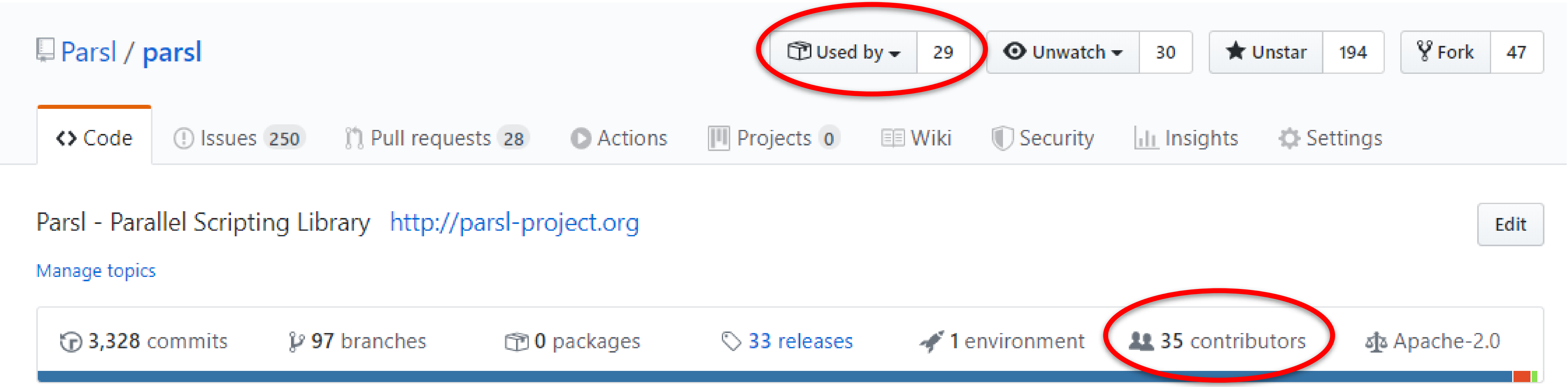


F

G


parsl

Parsl is an open-source Python community

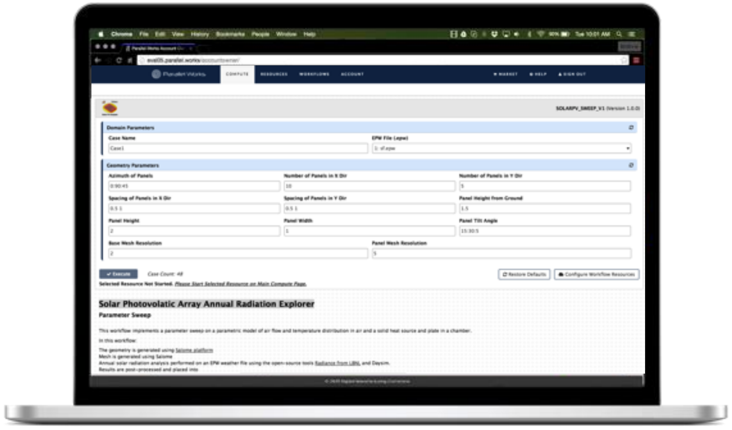


Supercharge your big compute problems with high-performance computing in the cloud.

Run compute-intensive simulation, modeling and data analytics workflows faster, at greater scale, and more cost effectively than ever before.

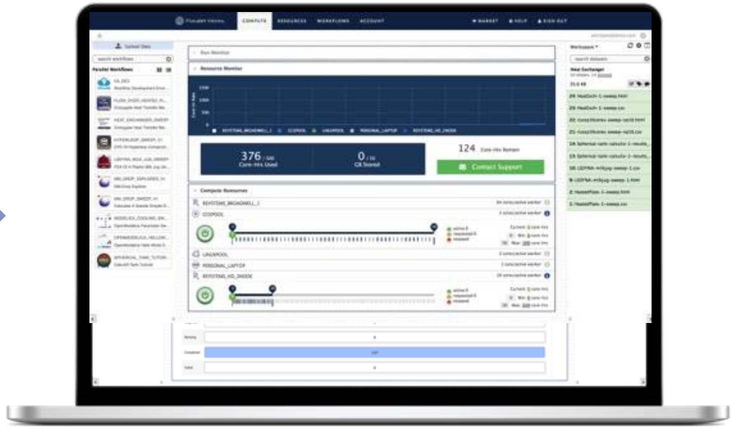
Parallel Works hosts workflow for design exploration

Specify Parameters



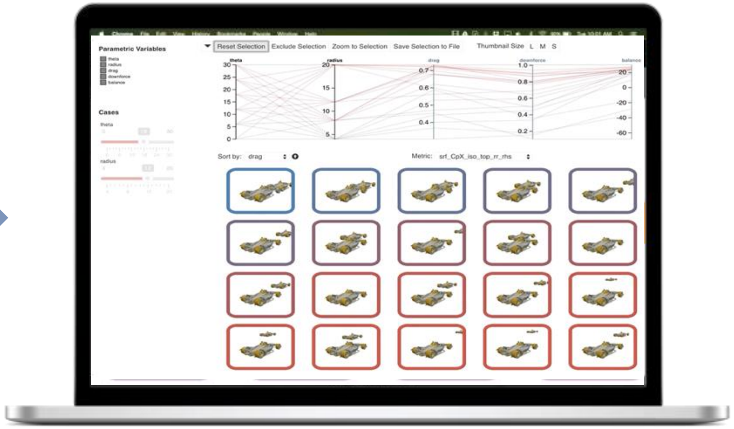
Specify inputs, parameters and variables

Run Parallel Workflow

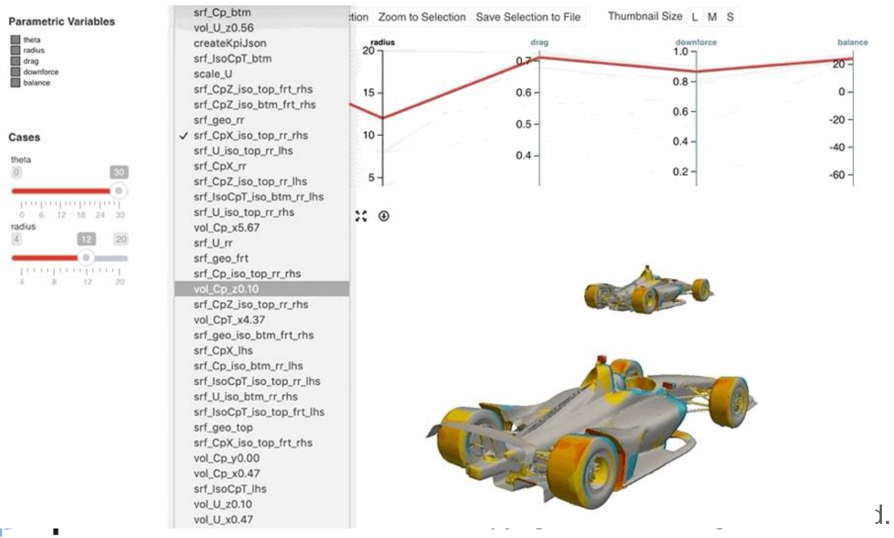


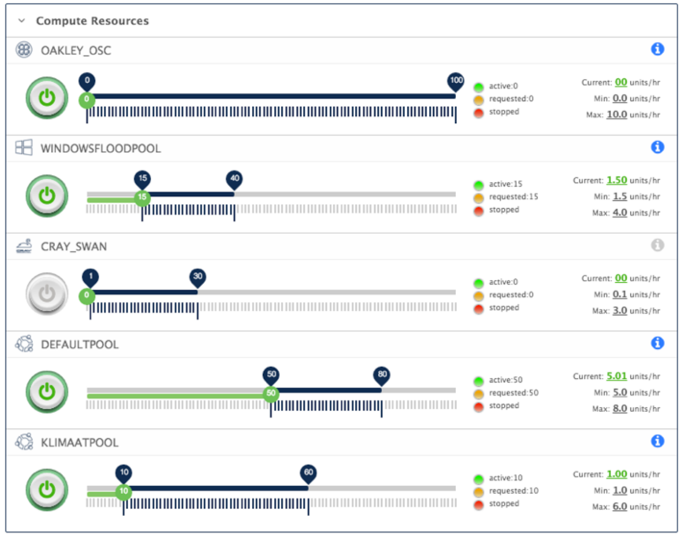
Track workflow progress and view intermediate results

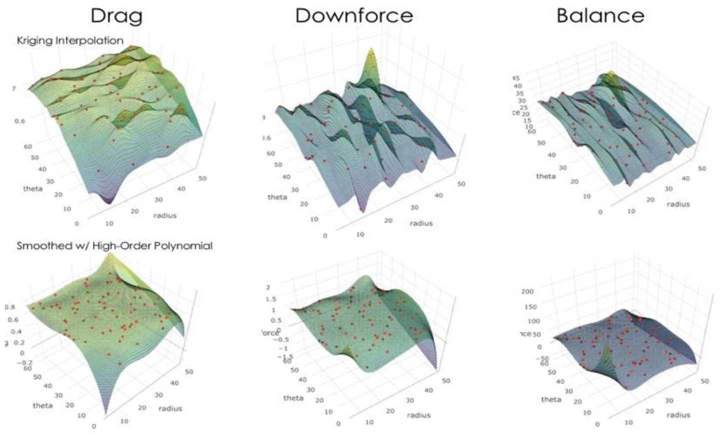
View Workflow Results



Rapidly analyze and visualize 1000x simulation results







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)

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

Scalable: efficient execution from laptops to the largest supercomputers

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

Questions?

http://parsl-project.org

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

amazon web services

U.S. DEPARTMENT OF ENERGY

NSF

THE UNIVERSITY OF CHICAGO

Argonne NATIONAL LABORATORY

ILLINOIS UNIVERSITY OF ILLINOIS AT URBANA CHAMPAIGN