Welcome! Parsl and funcX Fest 2021

Ian Foster, Daniel S. Katz, Kyle Chard

October 27-28, 2021

Parsl Code of Conduct

In the interest of fostering an open and welcoming environment, we as contributors and maintainers pledge to making participation in our project and our community a harassment-free and bullying-free experience for everyone, regardless of age, body size, disability, ethnicity, sex characteristics, gender identity and expression, level of experience, education, socio-economic status, nationality, personal appearance, race, religion, or sexual identity and orientation.

Examples of behavior that contributes to creating a positive environment include:

- Using welcoming and inclusive language
- Being respectful of differing viewpoints and experiences
- Gracefully accepting constructive criticism
- Focusing on what is best for the community
- Showing empathy towards other community members
- Respecting the work of others by recognizing acknowledgment/citation requests of original authors
- Being explicit about how we want our own work to be cited or acknowledged

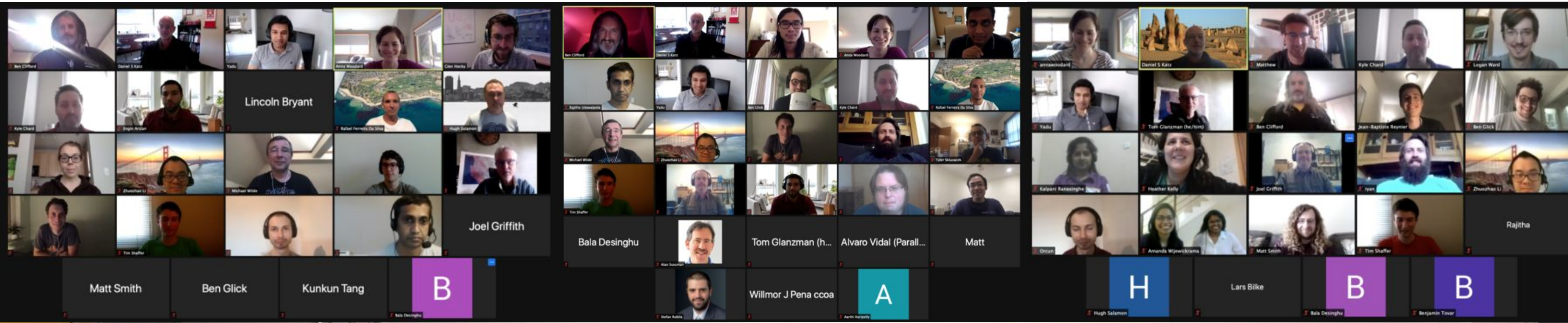
This meeting will follow the same Code of Conduct.

Issues: contact Dan Katz (dskatz@illinois.edu)

https://github.com/Parsl/parsl/blob/master/CODE_OF_CONDUCT.md

Parsl

funcX



Lincoln Bryant
Joel Griffith
Matt Smith
Ben Glick
Kunkun Tang
B
Bala Desinghu
Tom Glanzman (h...
Alvaro Vidal (Parall...
Matt
Willmor J Pena ccoa
A
Rajitha
H
Lars Bilke
B
B

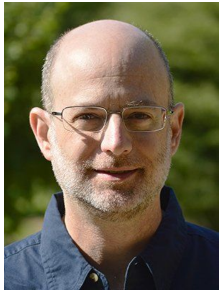
Introducing the team(s)










Uriel Mandujano



Thank you funding agencies and project partners

Parsl

1550588 (U Chicago/UIUC)
1550476 (Notre Dame),
1550475 (Colorado State)
1550562 (Northern Arizona)
1550528 (College of New Jersey)

NSF

funcX

2004894 (U Chicago)
2004932 (UIUC)

Argonne LDRDs

- 2022-0230 Productive Exascale Analysis Workflows for Numerical Cosmology
- 2021-0152 Creating a Robust and Scalable Framework for On-demand Analysis and AI-based Experiment Steering
- 2019-0217 Establishing a Usable, Scalable, and Reproducible Computational Ecosystem for Dark Energy Science

Dark Energy Science Collaboration

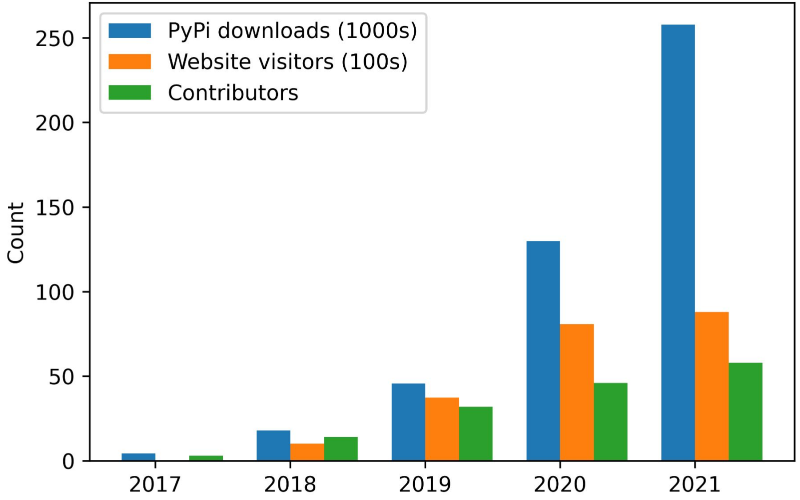
DOE ECP PRJ1008564 ExaWorks project

DOE DE-NA0003963 Center for Exascale-enabled Scramjet Design (CEESD)

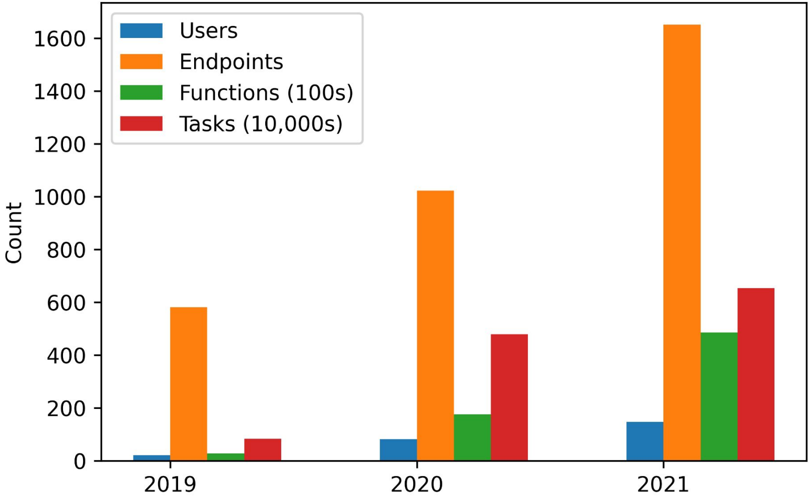
Discovery Partners Institute (DPI): Airborne-Satellite-AI-HPC integrative framework (ASAI)

U.S. DEPARTMENT OF ENERGY

Parsl and funcX are growing!



58 contributors, >400K PyPI downloads



> 10M tasks, >60K functions, >3000 endpoints

Goals for this meeting

- Learn about Parsl and funcX, and where they are going
- Learn about users of Parsl & funcX
  - Meet the community
  - Share experiences
- Find out how to contribute to Parsl/funcX
  - Help us develop and better engage the Parsl & funcX community
- Provide feedback to the Parsl/funcX team
  - Help us prioritize development activities
  - Help us identify shortcomings
  - Understand what needs work
- Form new collaborations

Agenda

Day 1

9:00 am - Welcome!

9:10 am - Intro to Parsl and funcX

9:30 am - Session 1 (Chair: Ben Clifford)

10:30 am - Tech talk: Zhuozhao Li, Parsl + funcX

10:45 am - Break

11:15 am - Session 2 (Chair: Dan Katz)

12:15 - Parallel Works Tech Talks

12:30 - Tech talk: Kir Nagaitsev, Asynchronous APIs in funcX

12:45 - Day 1 Closing

Day 2

1:00 pm - Session 3 (Chair: Yadu Babuji)

2:15 pm - Tech Talk: Douglas Thain, Resource Management for Dynamic Function Distribution

2:30 pm - Break

3:00 pm - Tech Talks: Ben Clifford, Ben Galewsky, and Raf Vescovi

4:00 Session 4 (Chair: Ryan Chard)

5:00 pm - Closing

https://parsl-project.org/parslfest2021.html

Introduction to Parsl and funcX

Kyle Chard
chard@uchicago.edu

Composition and parallelism

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/environment

- 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 platforms

funcX enables fire-and-forget remote and distributed execution

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 data dependencies. Natural parallel programming!

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

```
pip install parsl
```

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

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

```
Hello World!
```

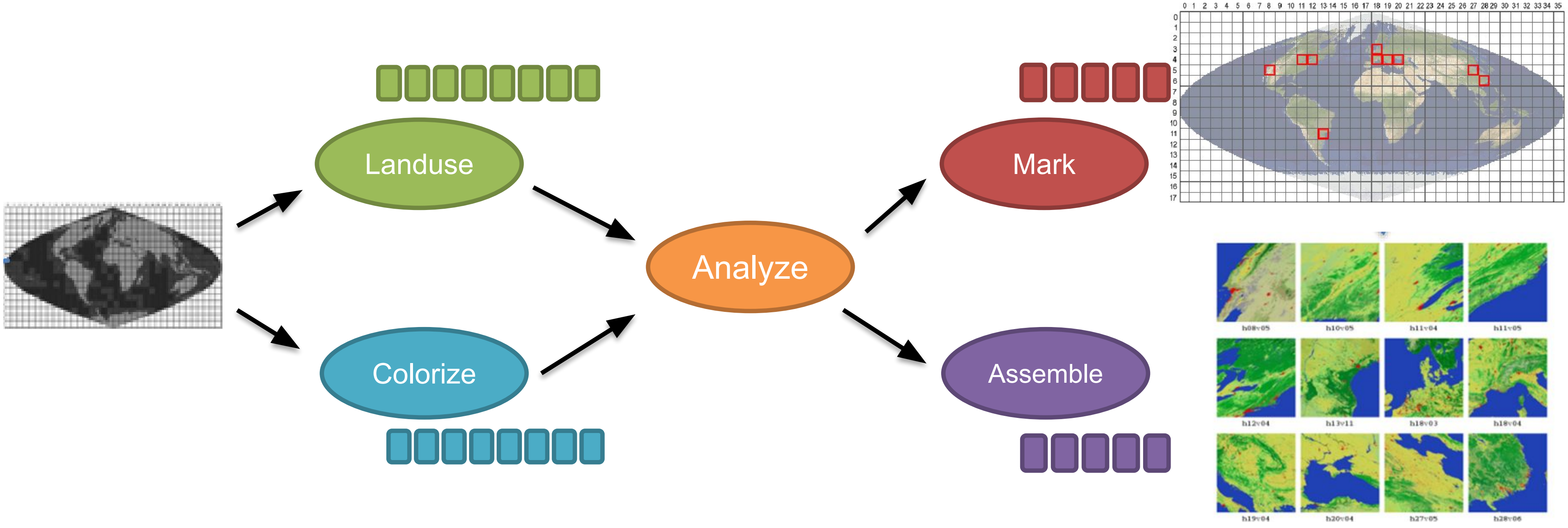
```python
@bash_app
def echo_hello(stdout='echo-hello.stdout'):
    return 'echo "Hello World!"'

echo_hello().result()

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

```
Hello World!
```

Parsl

Try Parsl: https://parsl-project.org/binder

Data-driven example: parallel geospatial analysis



Land-use Image processing pipeline for the MODIS remote sensor

Parsl decomposes parallel execution into a dynamic 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.

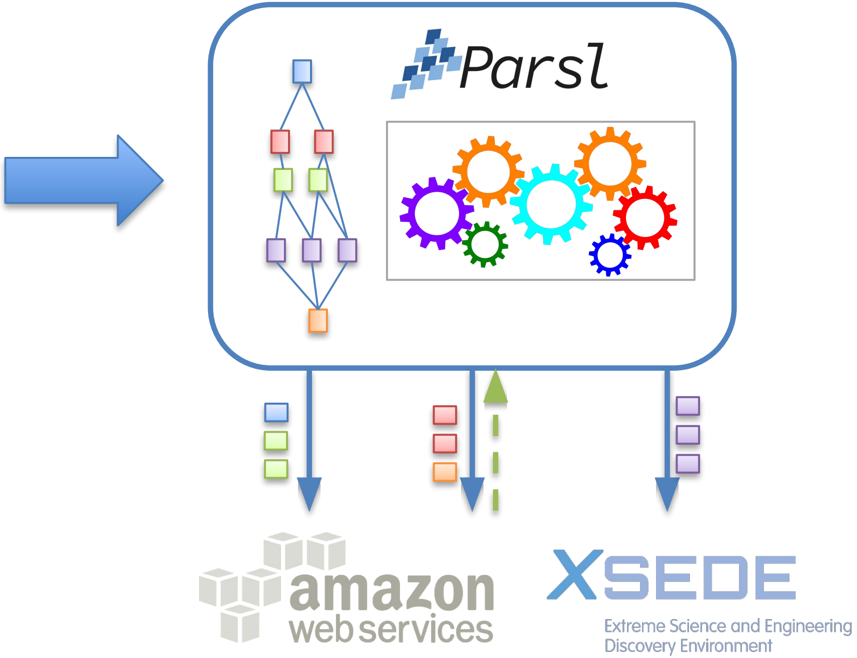
Each call to the function `pi()` is executed independently and in parallel. The `avg_three()` app is used to compute the average of the futures that were returned from the `pi()` calls.

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

Parsl

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

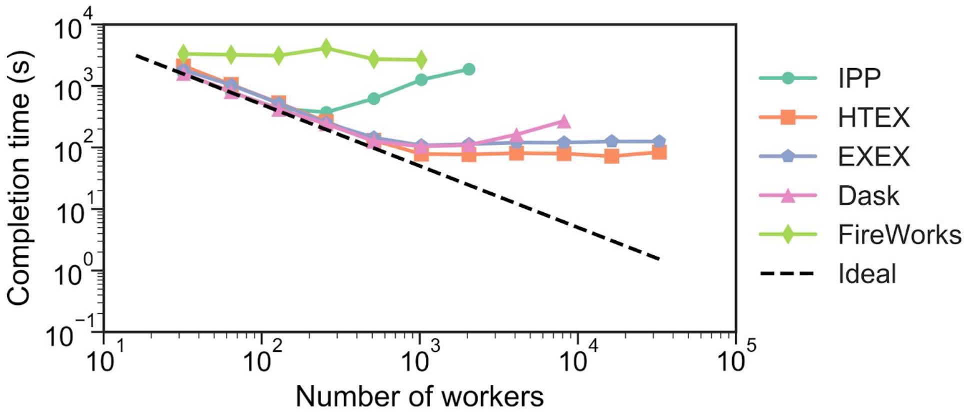
HTEX and EXEX outperform other Python-based approaches

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

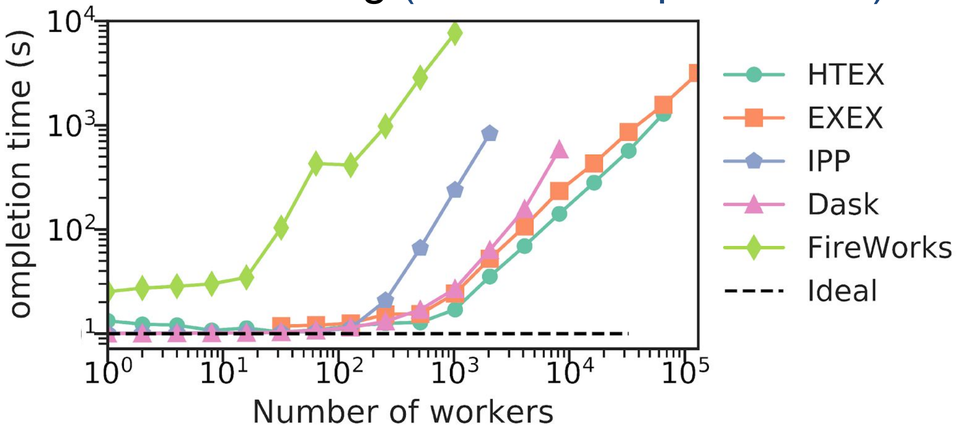
Framework	Maximum # of workers†	Maximum # of nodes†	Maximum tasks/second‡
Parsl-IPP	2048	64	330
Parsl-HTEX	65 536	2048*	1181
Parsl-EXEX	262 144	8192*	1176
FireWorks	1024	32	4
Dask distributed	4096	128	2617

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)



funcX: managed and federated FaaS

- Using Parsl to manage remote (and multi-site) computation can be difficult (e.g., persistent process, SSH connections, 2FA)
- Many Parsl programs have few (or no dependencies)
- Configuring Parsl for different systems can be complicated

- Can we build a simpler model for running tasks remotely?
  - Cloud-hosted service offering fire-and-forget function execution
  - Register and share FaaS compute endpoints
  - Register and share Python functions
  - Reliable, scalable, secure function execution on arbitrary remote endpoints

funcX

Try funcX: https://funcx.org/binder

Transform laptops, clusters, clouds into function serving endpoints

- Python-based agent (pip or Conda) installable in user space
- Elastically provisions resources from local, cluster, kubernetes, or cloud system (using Parsl)
- Manages concurrent execution on provisioned resources
- Optionally manages execution in containers
- Share endpoints with collaborators

```
$ pip install funcx-endpoint

$ funcx-endpoint configure myep

$ funcx-endpoint start myep
```

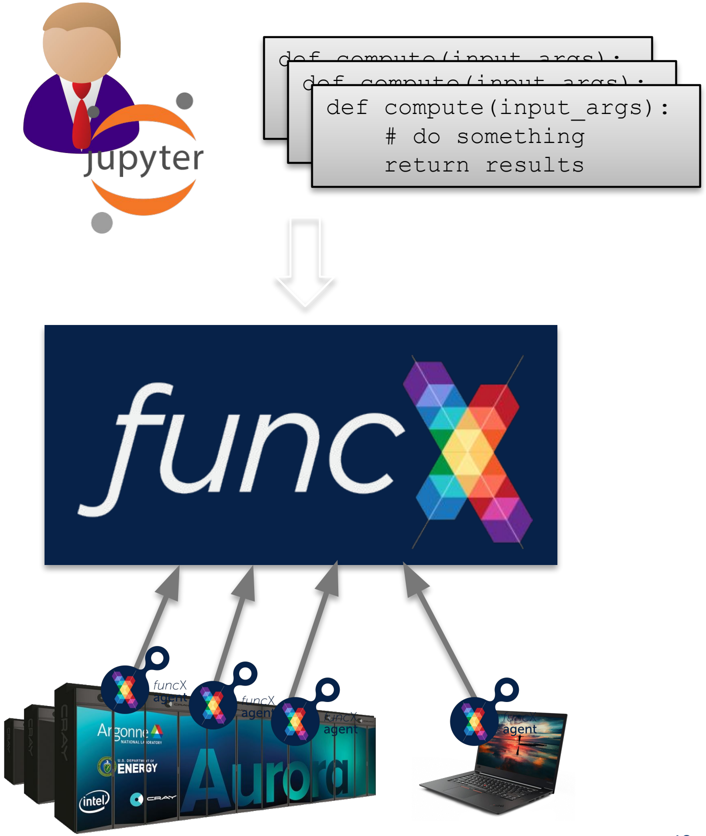
Register and share functions

Create funcX client (and authenticate)

```python
from funcx.sdk.client import FuncXClient

fxc = FuncXClient()
```

Define and register Python function

```python
def hello_world():
    return "Hello World!"

func_uuid = fxc.register_function(hello_world)
print(func_uuid)
```



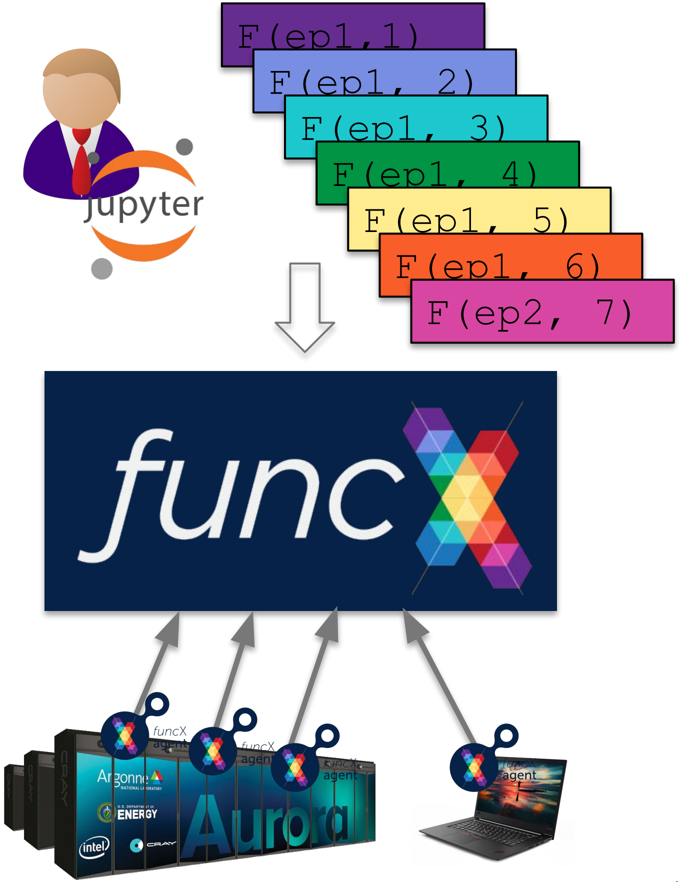
Execute tasks on any accessible endpoint

Select: function ID, endpoint ID, and input arguments

```
tutorial_endpoint = '4b116d3c-1703-4f8f-9f6f-39921e5864df'
res = fxc.run(endpoint_id=tutorial_endpoint,
              function_id=func_uuid,
              arg1, arg2, arg3)
```

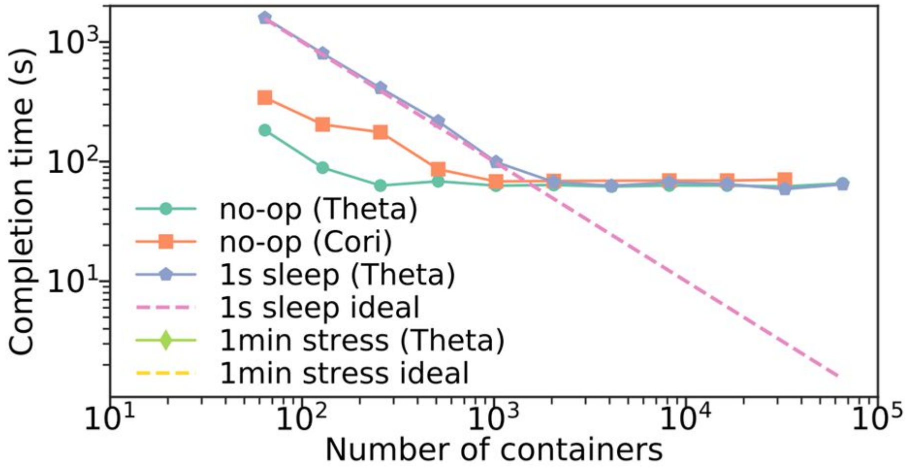
Retrieve results asynchronously (funcX stores results in the cloud)

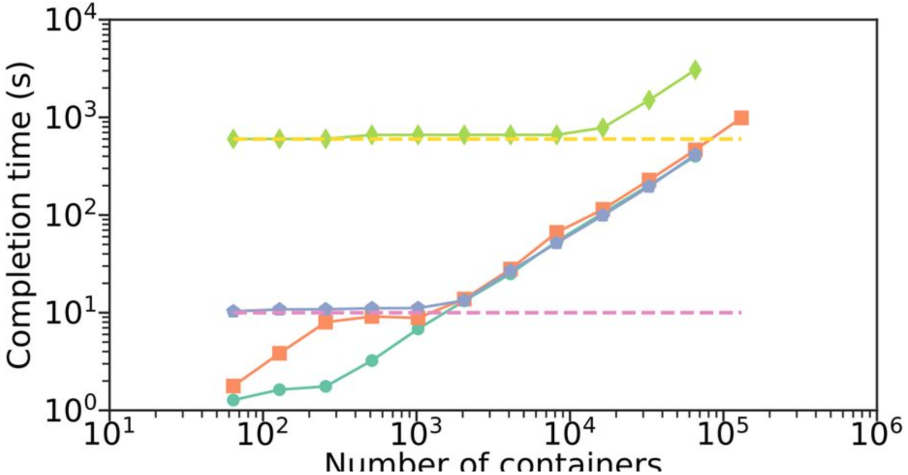
```
print(fxc.get_result(res))
```



funcX scales to 100K+ workers

- funcX endpoints deployed on ALCF Theta and NERSC Cori
- Strong scaling (100K concurrent functions) shows good scaling up to 2K containers even with short no-op/sleep tasks
- Weak scaling (10 tasks per container) scales to 131K concurrent containers (1.3M tasks)

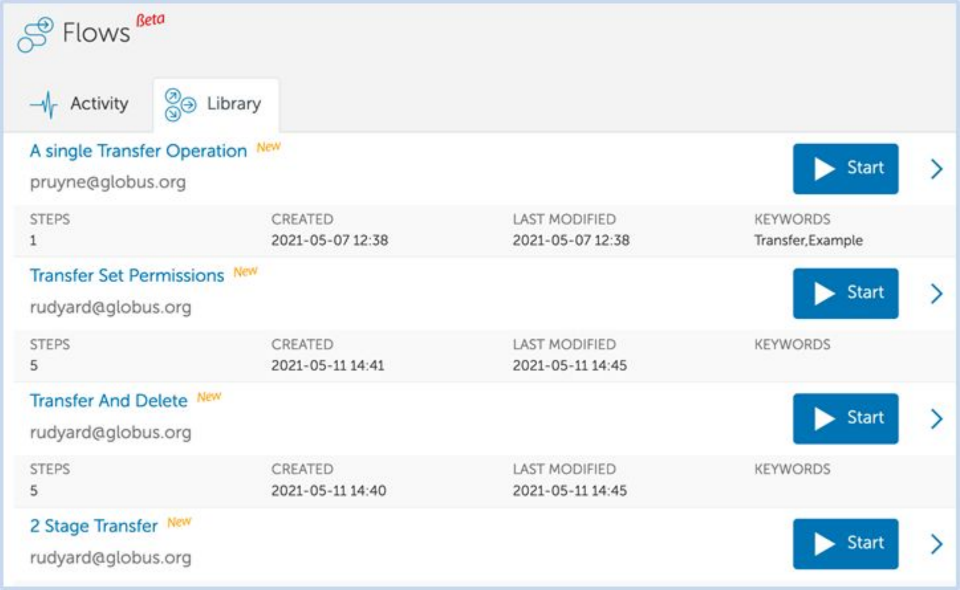


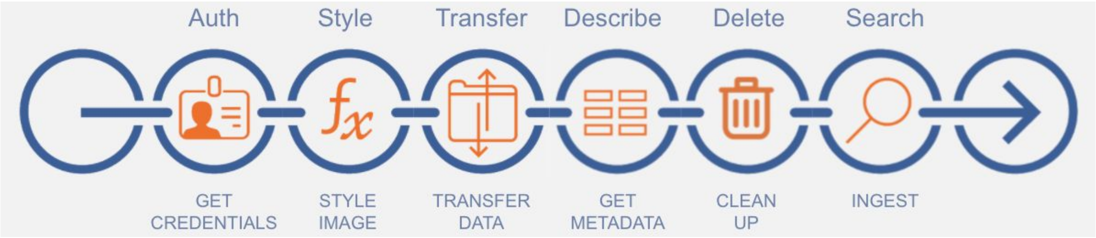


R. Chard et.al. "FuncX: A Federated Function Serving Fabric for Science."
ACM International Symposium on High-Performance Parallel and Distributed Computing (HPDC). 2020.

Automating the research lifecycle with the Globus Automate platform and funcX

- Managed, secure, and reliable task orchestration across heterogenous resources
- Declarative language for composition
- Extensible custom actions
- Event-driven execution





When should you use Parsl or funcX?

Parsl

Workflows

Single site

High performance

Management of MPI apps

Integrated wide-area data management

funcX

Bag of tasks

One or more sites

Fire-and-forget execution

Execution in containers

Share functions and endpoints

Automated, event-based computing

Parsl + funcX

Workflows executed remotely across one or more sites

Agenda

Day 1

9:00 am - Welcome!

9:10 am - Intro to Parsl and funcX

9:30 am - Session 1 (Chair: Ben Clifford)

10:30 am - Tech talk: Zhuozhao Li, Parsl + funcX

10:45 am - Break

11:15 am - Session 2 (Chair: Dan Katz)

12:15 - Parallel Works Tech Talks

12:30 - Tech talk: Kir Nagaitsev, Asynchronous APIs in funcX

12:45 - Day 1 Closing

Day 2

1:00 pm - Session 3 (Chair: Yadu Babuji)

2:15 pm - Tech Talk: Douglas Thain, Resource Management for Dynamic Function Distribution

2:30 pm - Break

3:00 pm - Tech Talks: Ben Clifford, Ben Galewsky, and Raf Vescovi

4:00 Session 4 (Chair: Ryan Chard)

5:00 pm - Closing

https://parsl-project.org/parslfest2021.html

Parsl & funcX Fest 2021

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

Introducing the team(s)

Rachana Ananthakrishnan
Yadu Babuji
Ben Blaiszik
Josh Bryan
Kyle Chard
Ryan Chard
Ben Clifford
Ian Foster
Ben Galewsky
Dan Katz
Zhuozhao Li
Uriel Mandujano
Kir Nagaitsev
Stephen Rosen
Tyler Skluzacek
Logan Ward
Mike Wilde
Anna Woodard

Expressing parallelism using Parsl

1) Wrap the science applications as Parsl Apps:

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

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

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

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]])
```

FuncX: a federated function serving ecosystem for research

Endpoints:

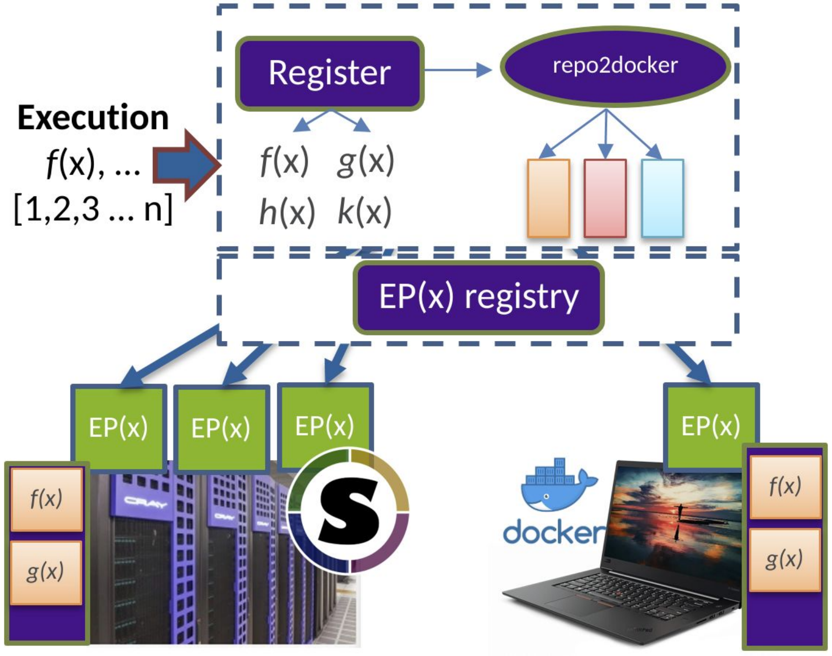
- User-deployed and managed
- Dynamically provision resources, deploy containers, and execute functions
- Exploit local architecture/accelerators

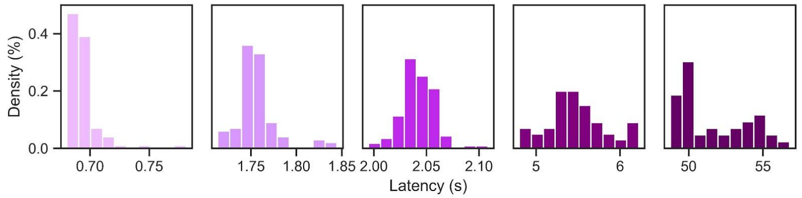
funcX Service:

- Single reliable cloud interface
- Register and share endpoints
- Register, share, run functions

Choose where to execute functions

- Closest, cheapest, fastest, accelerators …



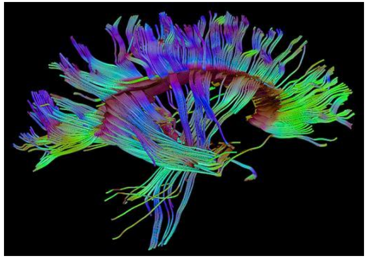


(a) tabular file extraction (b) MNIST digit prediction (c) DIALS stills process (d) tomographic preview (e) correlation spectroscopy

Parallel applications require different execution models

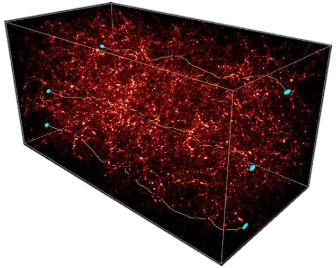
High-throughput workloads

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



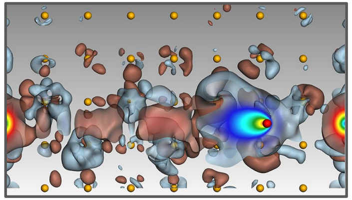
Extreme-scale workloads

- 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

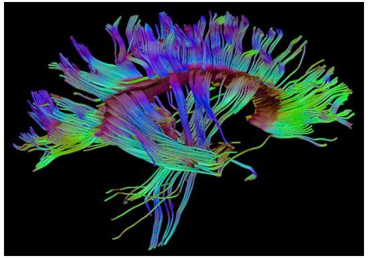
- Materials science, cosmic ray shower analysis, machine learning inference
- **Requirements**: 10s of nodes, seconds-minutes, rapid response, pipelining



Parsl implements a modular executor interface

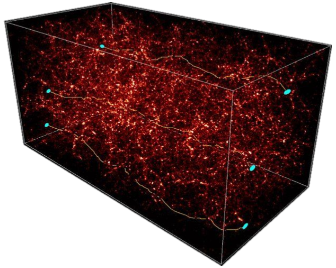
High-throughput executor (HTEX)

- 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



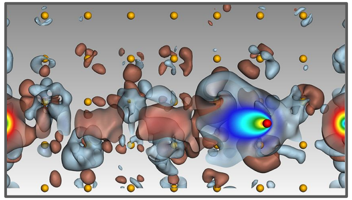
Extreme-scale executor (EXEX)*

- 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

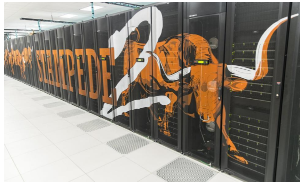


Others: WorkQueue and IPyParallel

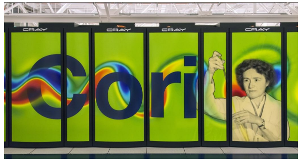
Parsl scripts are execution provider independent

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

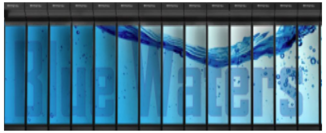
Growing support for various schedulers and cloud vendors



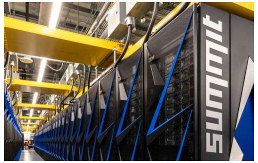
Open Science Grid



amazon web services



Google Cloud



XSEDE

⊟ 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

Separation of code and execution

sample_configs.py

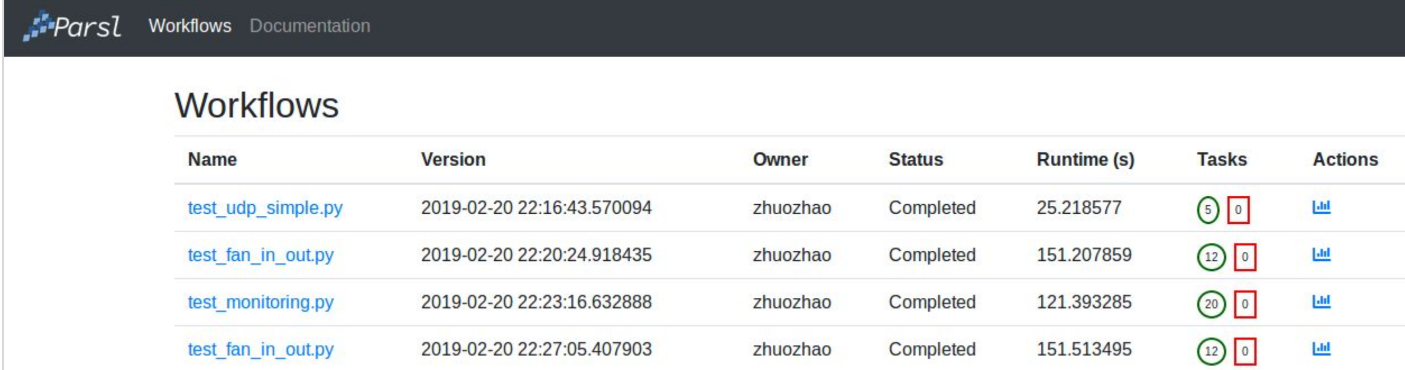
```python
# ... imports

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...
```

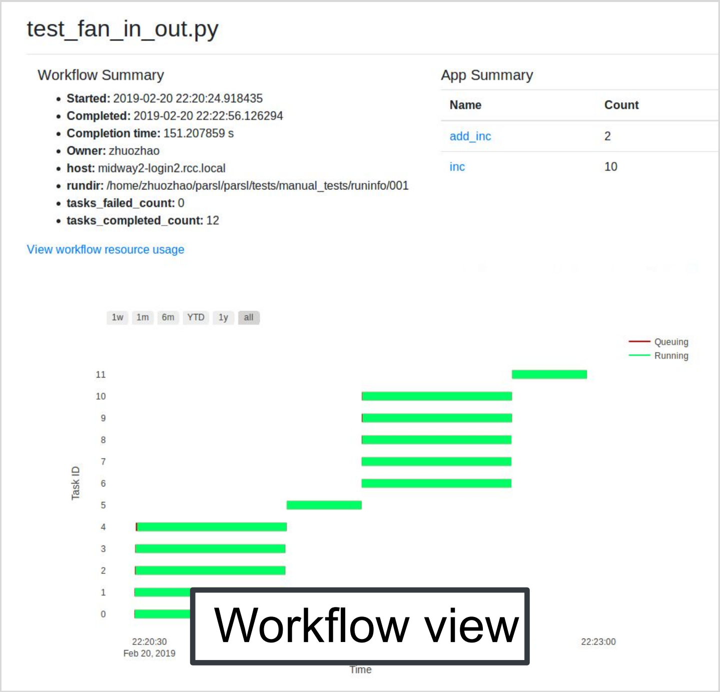
Choose execution environment at runtime. Parsl will direct tasks to the configured execution environment(s).

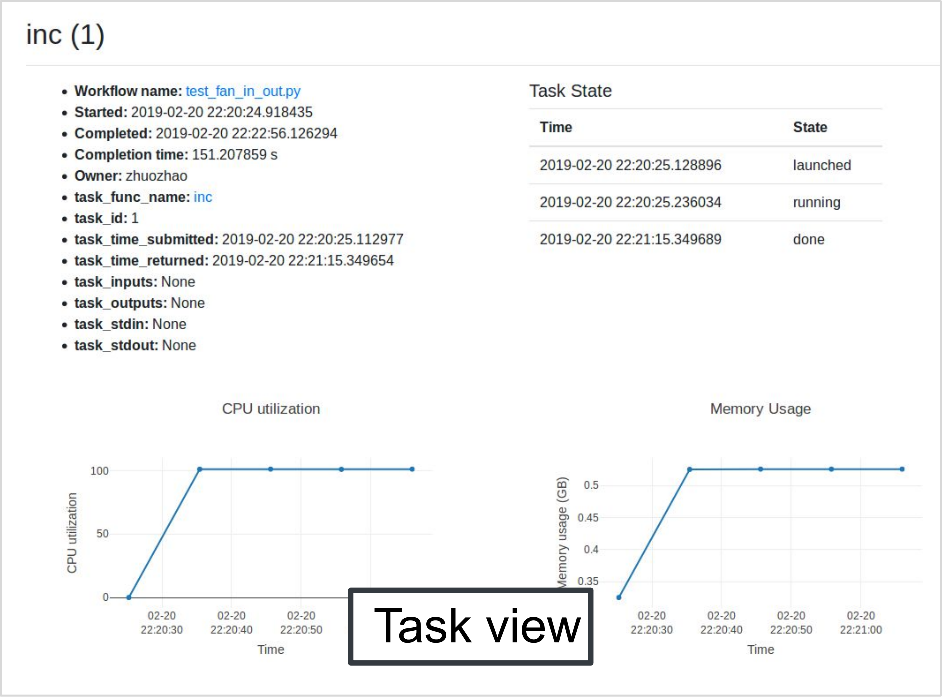
Monitoring and visualization


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	

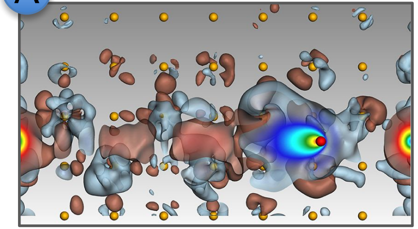




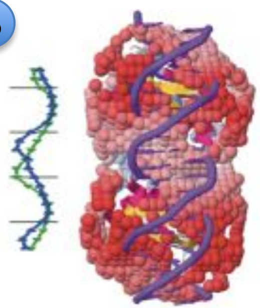
Parsl is being used in a wide range of scientific applications

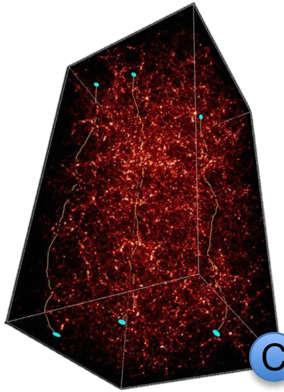
- A Machine learning to predict stopping power in materials
- B Protein and biomolecule structure and interaction
- C LSST simulation and weak lensing using sky surveys
- D Cosmic ray showers in QuarkNet
- E Information extraction to classify image types in papers
- F Materials science at the Advanced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

A



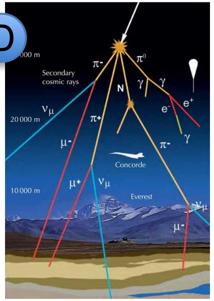
B



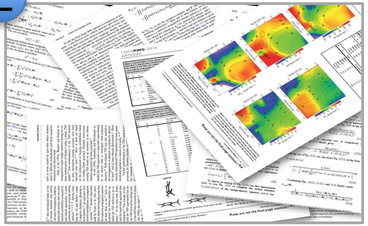


C

D



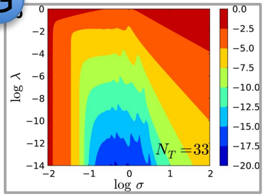
E



F

Red indicates higher statistical confidence in data

G



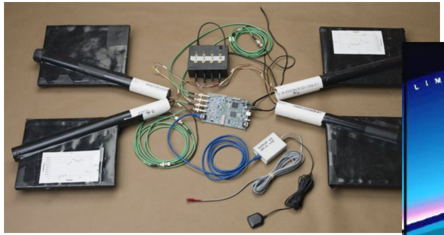
HPC and science education using Parsl

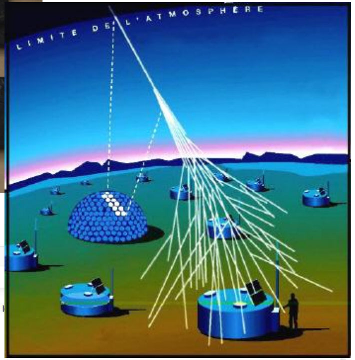
HPC and Samoan Fire Knife Dancing, What Could Go Wrong?

Ben Glick shares a rich undergraduate experience that ranges from building an HPC system to dancing with fire.

Posted by @vsoch · 1 min read

http://us-rse.org/rse-stories/2020/ben-glick/





jupyter 5_Flux_script (unsaved changes)

File Edit View Insert Cell Kernel Widgets Help

(remember, `wc -l` returns a count of the number of lines in the file). These l... them.

The Parsl Flux App

For convenience, we'll wrap the UNIX command-line invocation of the `Flux.pl` script in a Parsl App, which will make it easier to work with from within the Jupyter Notebook environment.

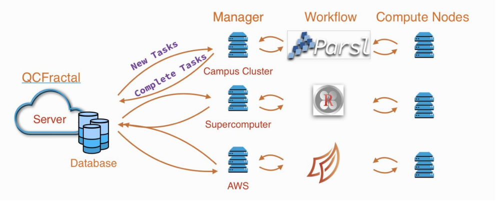
In [ ]:
```
# The prep work:
import parsl
from parsl.config import Config
from parsl.executors.threads import ThreadPoolExecutor
from parsl.app.app import bash_app,python_app
from parsl import File

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

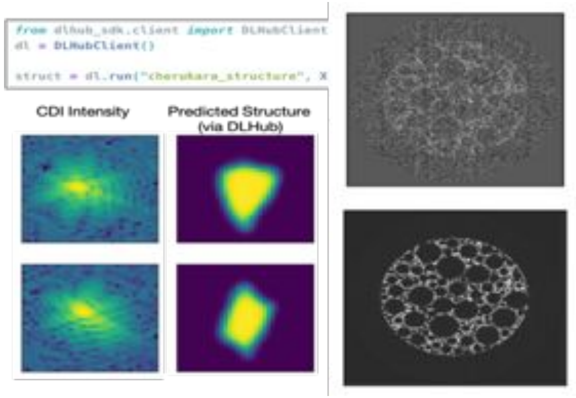
In [ ]:
```
# The App:
@bash_app
def Flux(inputs=[], outputs=[], binWidth='600', geoDir='geo/', stdout='stdout.txt', stderr='stderr.txt'):
    return 'perl ./perl/Flux.pl %s %s %s %s' % (inputs[0], outputs[0], binWidth, geoDir)
```

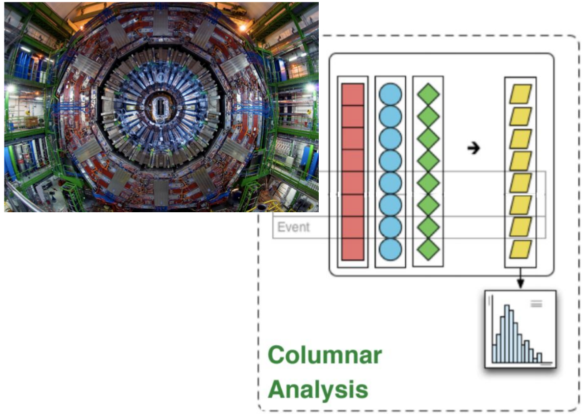
Edit stuff below to use the App

https://quarknet.org/content/about-e-labs

QCArchive
A MolSSI Project







QCArchive
Compile, aggregate, query, and share quantum chemistry data on diverse systems

Data and Learning Hub for Science (DLHub)
Interactive execution of user-provided machine learning models in real-time

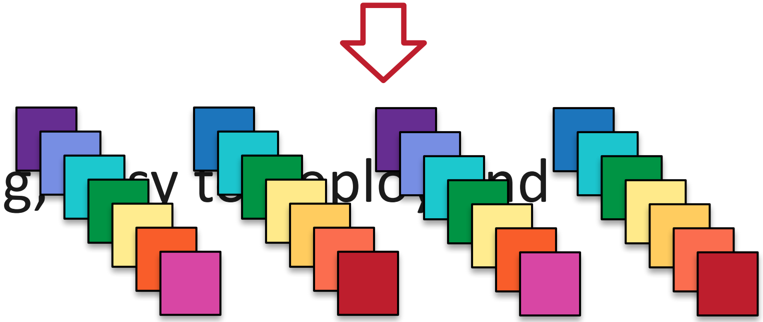
Coffea: Column Object Framework for Effective Analysis
Back-end-agnostic data processing libraries for granular event-based HEP analysis

Function as a Service (FaaS)

Serverless: Cloud provider provisions and manages all infrastructure

FaaS: Developers work in terms of progr

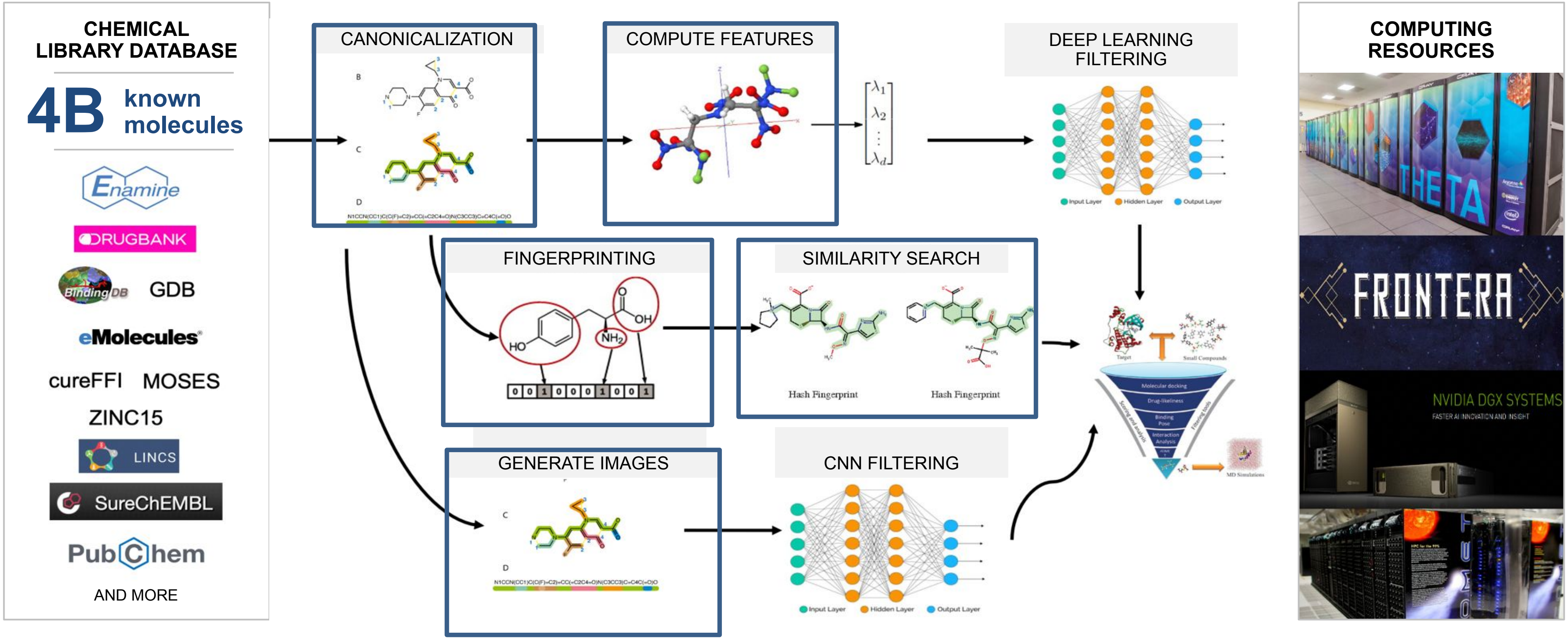
```
def compute(input_args):
    # do something
    return results
```



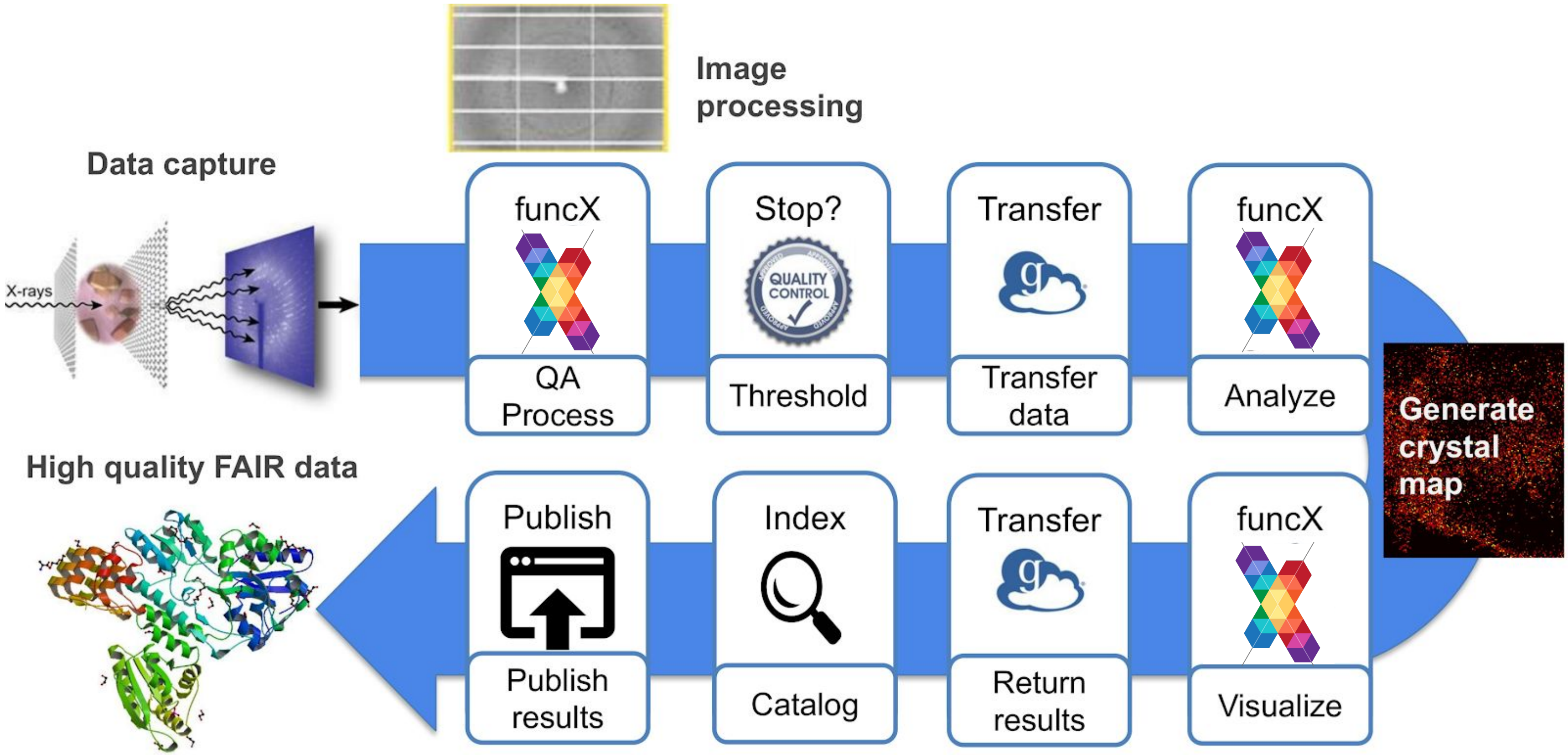
1. Pick a runtime (e.g., Python)
2. Register function code
3. Run (and scale)

Low latency, on-demand, elastic scaling,
update

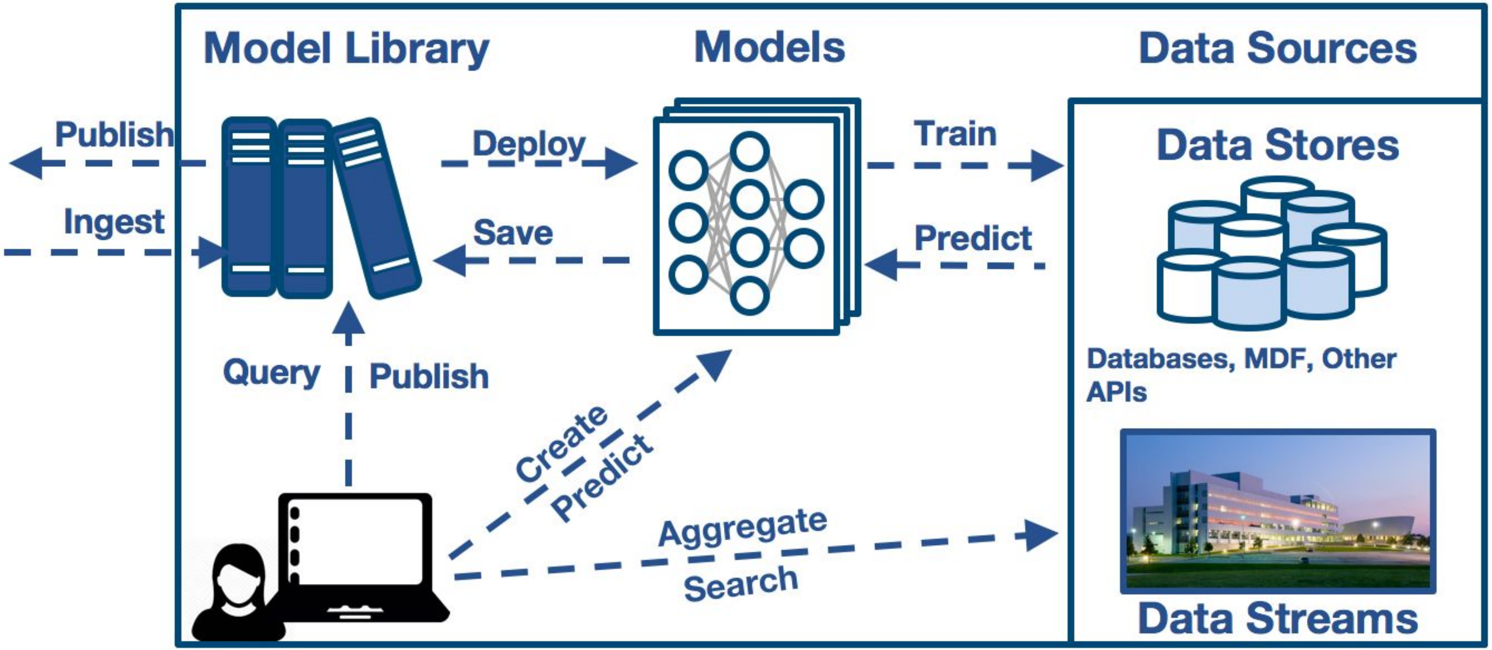
The COVID’19 data pipeline: Using AI and supercomputers to accelerate drug development

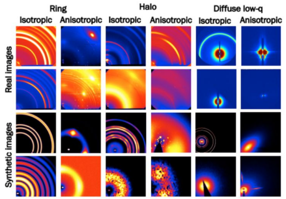


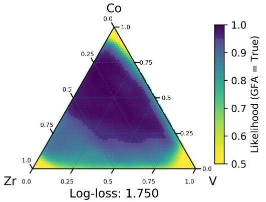
Research Automation: Serial Crystallography

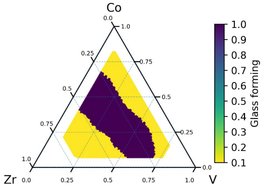


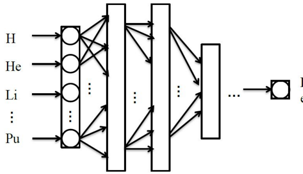
Data and Learning Hub for Science (DLHub)



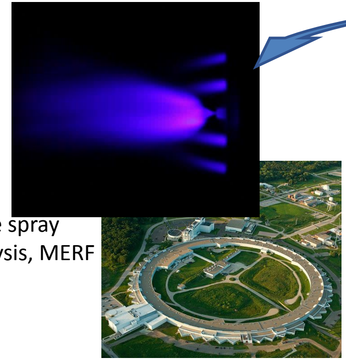






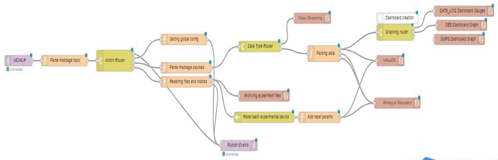


Manufacturing and machine learning



Flame spray pyrolysis, MERF

The Manufacturing and ML platform **(MDML)**



MINIO

influxdb

Grafana Real-Time Dashboards

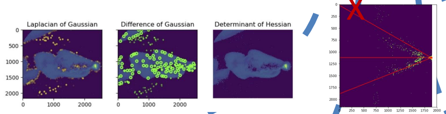
1. Instrument sensors stream data to the MDML

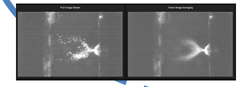
2. Use FaaS to analyze data on-demand

f(X)
func
X

3. FaaS tasks distributed across the computing continuum

4. Results are used to guide the experiment





Edge devices

Laboratory machines

HPC

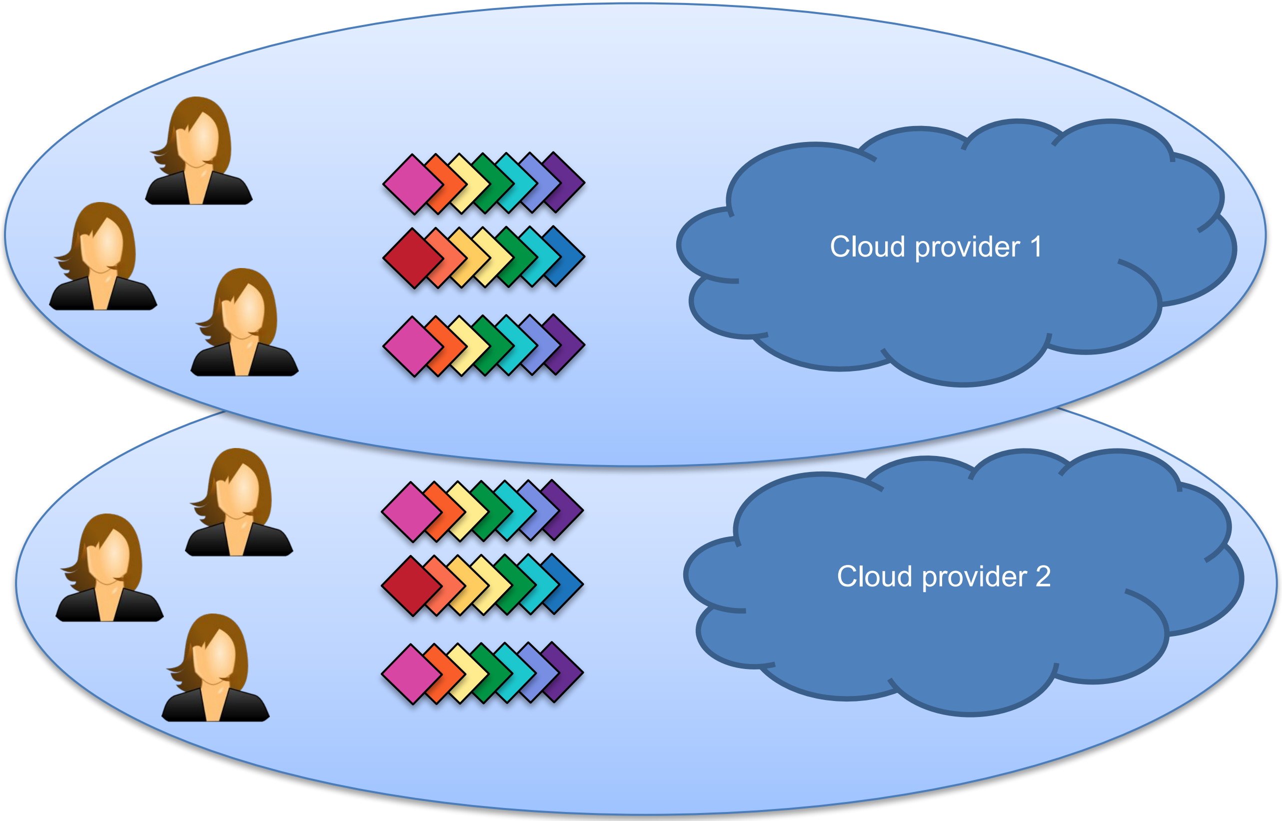
Compute and storage continuum

Lessons learned applying funcX to science use cases

- ✔ Abstracts the complexity of using diverse compute resources
- ✔ Simplicity: automatic scaling, single interface
- ✔ Flexible web-based authentication model
- ✔ Enables event-based processing and automated pipelines
- ✔ Increases portability between sites, systems, etc.
- ✔ Resources can be used efficiently and opportunistically
- ✔ Enables secure function/endpoint sharing with collaborators

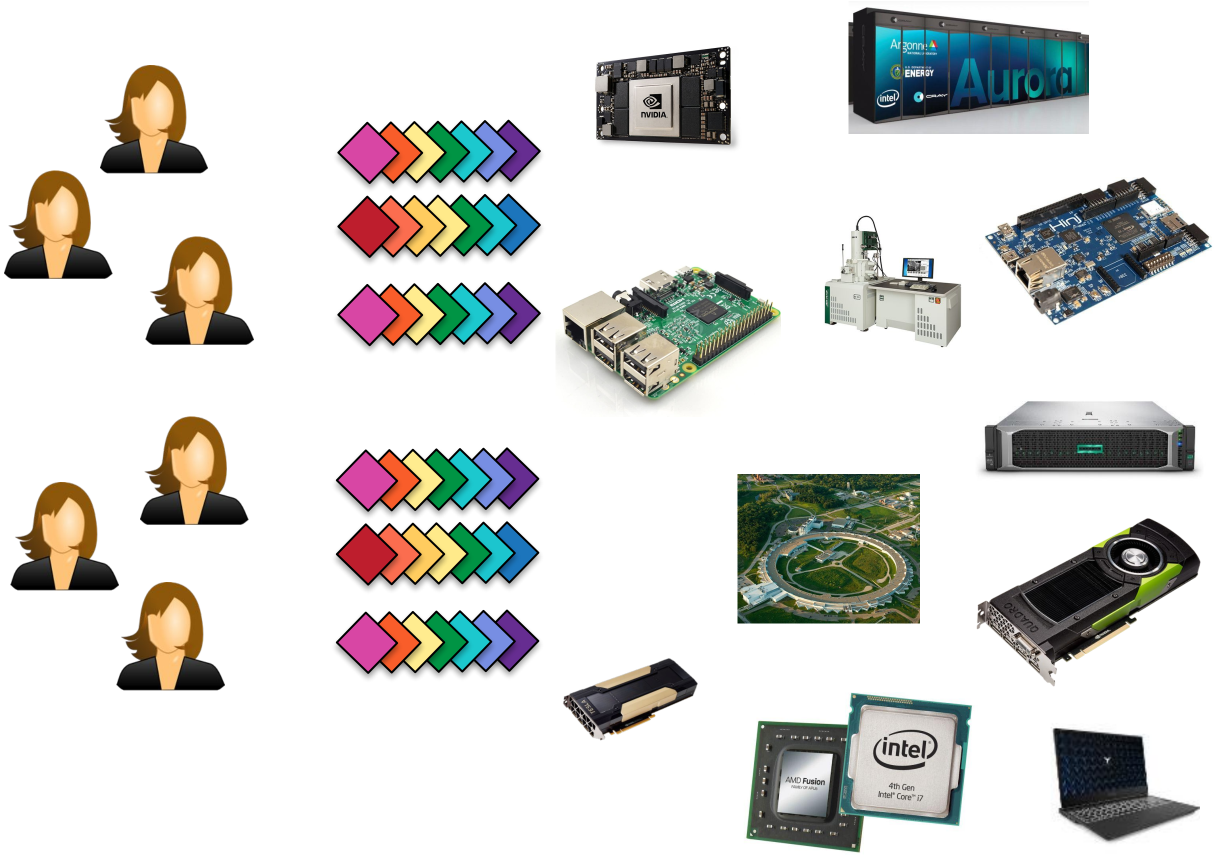
- FaaS is not suitable for some applications
- Ratio of data size to compute must be reasonable
- Containerization does not always provide entirely portable codes
- Coarse allocation models do not map well to fine grain/short functions
- Decomposing applications isn’t always easy (or possible)

FaaS as offered by cloud providers



- Single provider, single location to submit and manage tasks
- Homogenous execution environment
- Transparent and elastic execution
- Integrated with cloud provider data management

FaaS as an interface to the distributed computing ecosystem



We still want

- Single interface
- Homogenous execution environment
- Transparent and elastic execution
- Integrated with data management