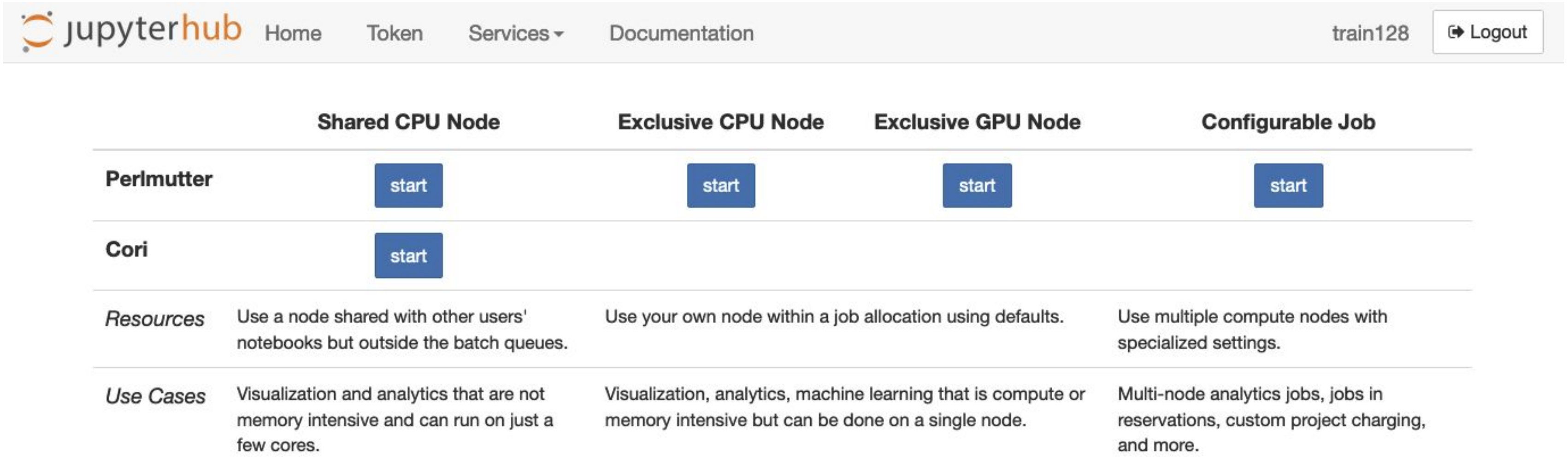
Parsl

https://parsl-project.org/

Bjoern Enders (NERSC) & Tyler Skluzacek (OLCF)

The Parsl demo at NERSC will be run via Jupyter

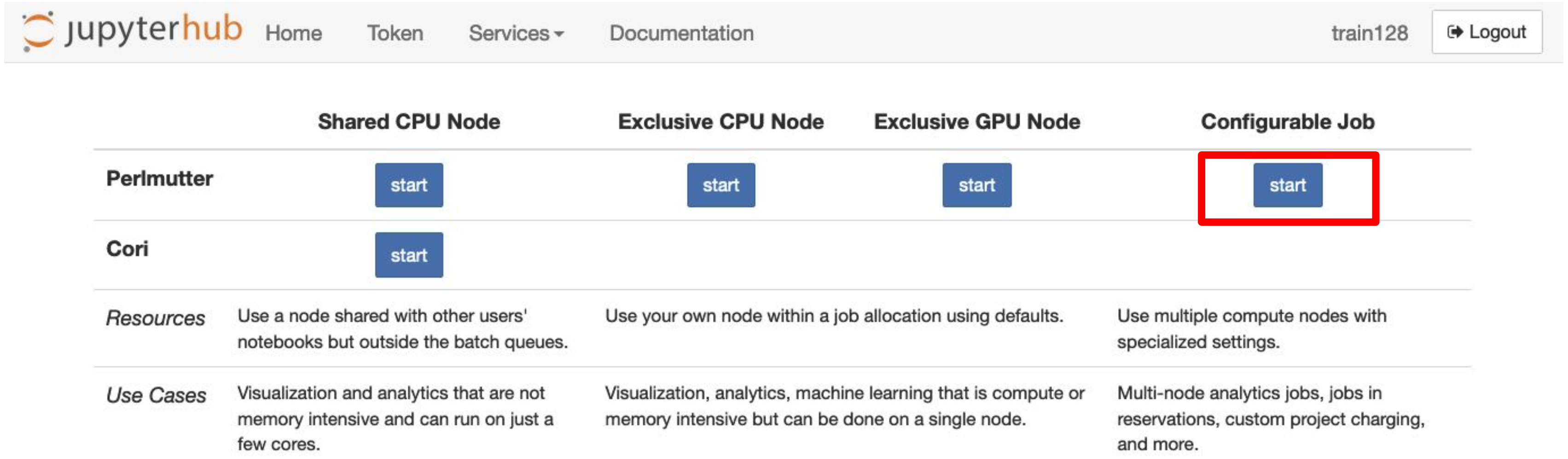
- Go to https://jupyter.nersc.gov
- Sign in using your training account credentials
- Select your preferred jupyter instance:



	Shared CPU Node	Exclusive CPU Node	Exclusive GPU Node	Configurable Job
Perlmutter	start	start	start	start
Cori	start			
Resources	Use a node shared with other users' notebooks but outside the batch queues.	Use your own node within a job allocation using defaults.		Use multiple compute nodes with specialized settings.
Use Cases	Visualization and analytics that are not memory intensive and can run on just a few cores.	Visualization, analytics, machine learning that is compute or memory intensive but can be done on a single node.		Multi-node analytics jobs, jobs in reservations, custom project charging, and more.

Using Reservations in Tutorials

- Go to https://jupyter.nersc.gov and select: "Configurable Job" in the "Perlmutter" row



	Shared CPU Node	Exclusive CPU Node	Exclusive GPU Node	Configurable Job
Perlmutter	start	start	start	start
Cori	start			
Resources	Use a node shared with other users' notebooks but outside the batch queues.	Use your own node within a job allocation using defaults.		Use multiple compute nodes with specialized settings.
Use Cases	Visualization and analytics that are not memory intensive and can run on just a few cores.	Visualization, analytics, machine learning that is compute or memory intensive but can be done on a single node.		Multi-node analytics jobs, jobs in reservations, custom project charging, and more.

Jupyter Options:

Leave defaults, except:

Constraint:

`cpu`

Account:

`ntrain7`

Reservation

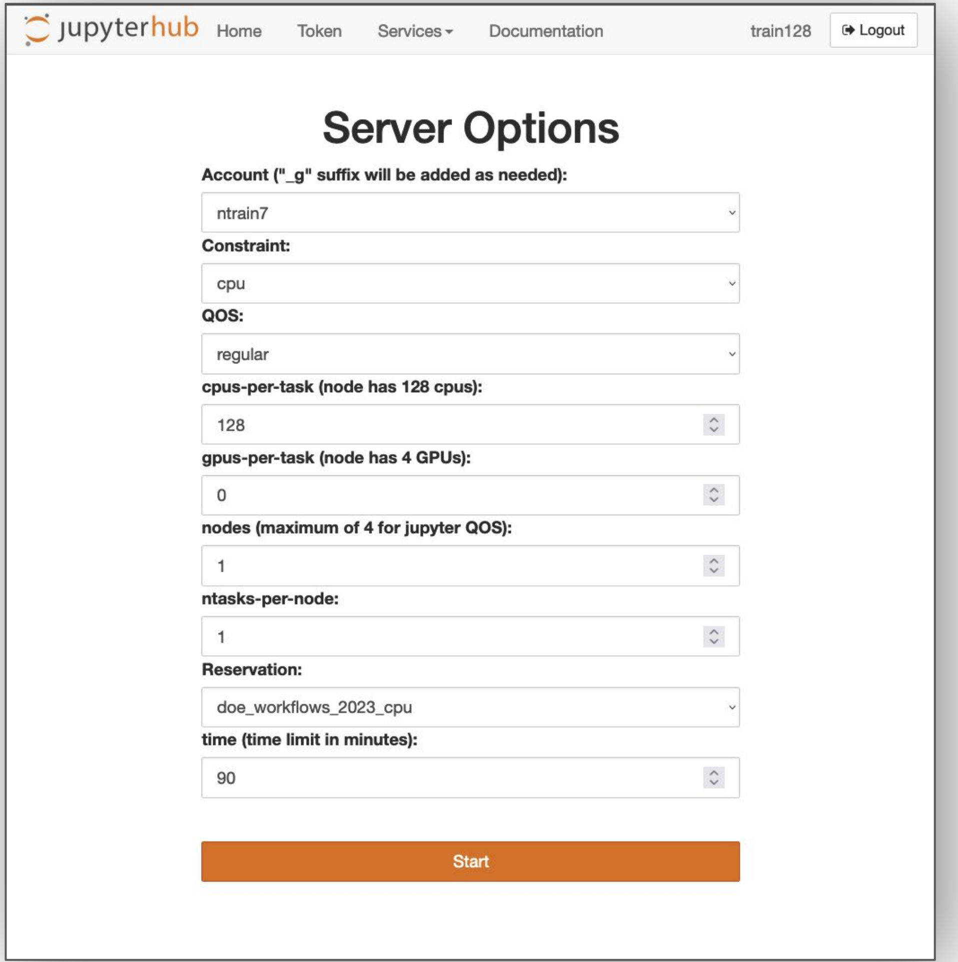
`doe_workflows_2023_cpu`

Time

`90`

QOS

`regular`



Server Options

Account ("_g" suffix will be added as needed): ntrain7

Constraint: cpu

QOS: regular

cpus-per-task (node has 128 cpus): 128

gpus-per-task (node has 4 GPUs): 0

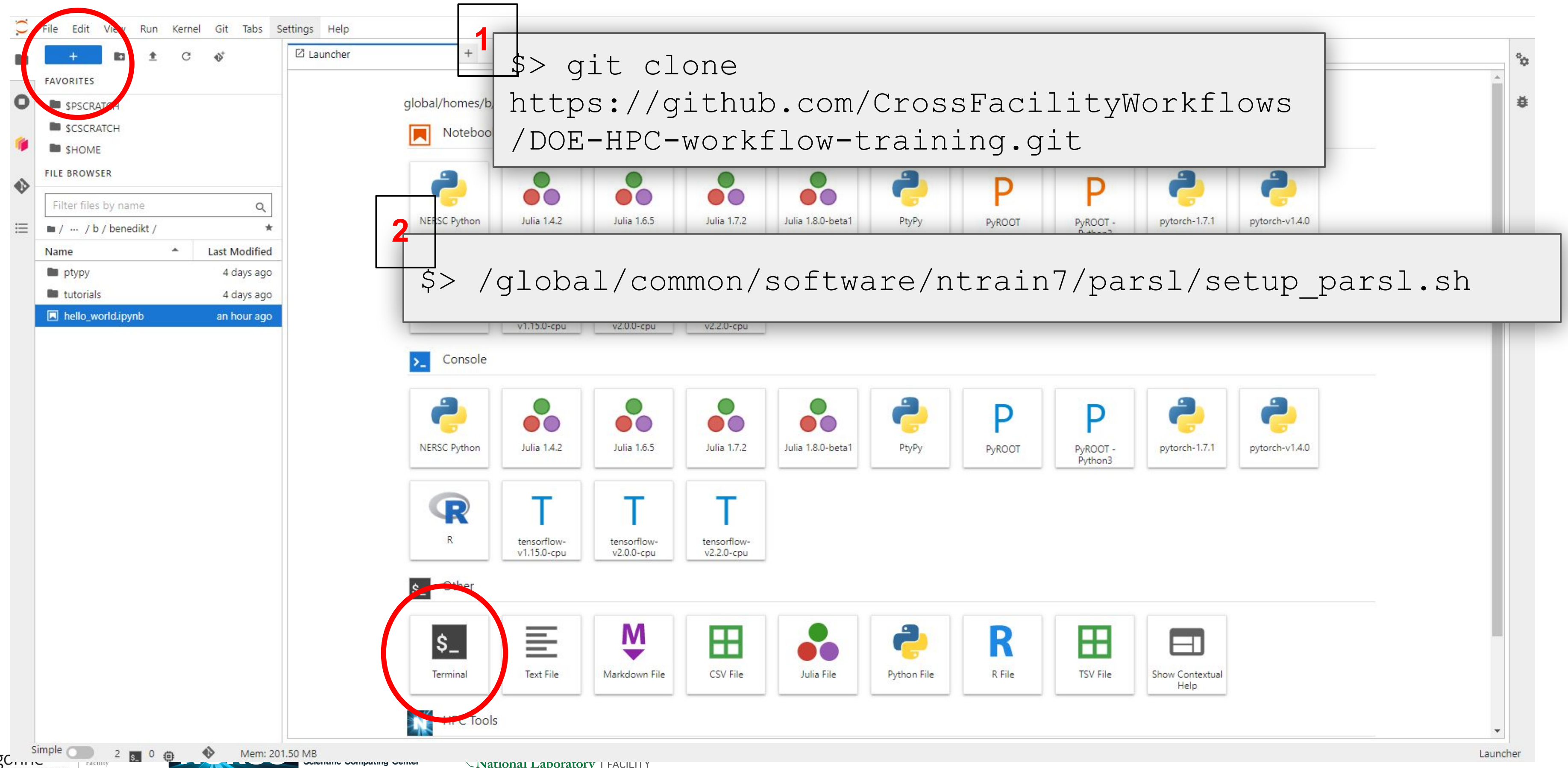
nodes (maximum of 4 for jupyter QOS): 1

ntasks-per-node: 1

Reservation: doe_workflows_2023_cpu

time (time limit in minutes): 90

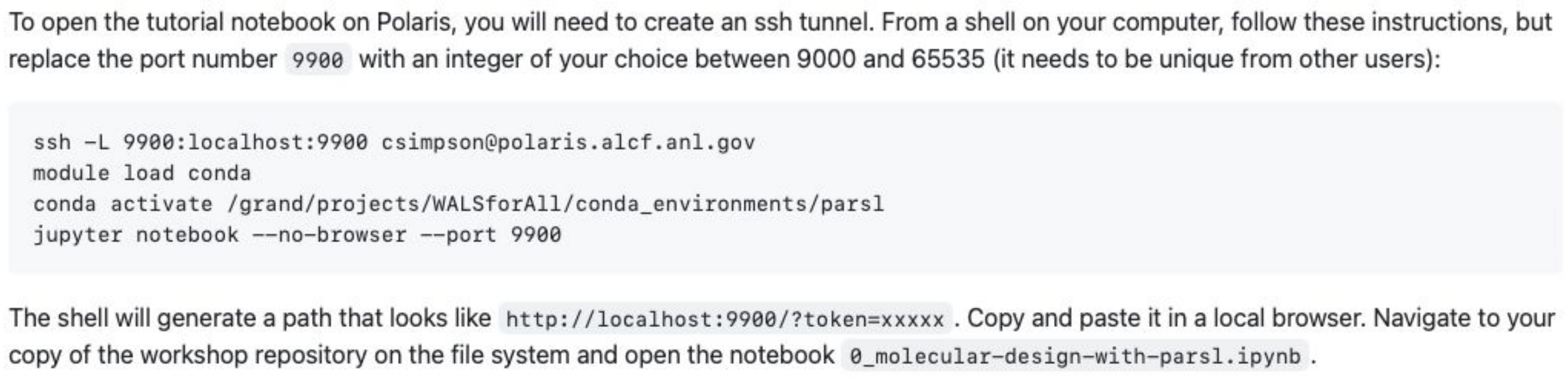
Open Terminal, download tutorials, setup



```
$> git clone
https://github.com/CrossFacilityWorkflows
/DOE-HPC-workflow-training.git
```

```
$> /global/common/software/ntrain7/parsl/setup_parsl.sh
```

Instructions for ALCF



To open the tutorial notebook on Polaris, you will need to create an ssh tunnel. From a shell on your computer, follow these instructions, but replace the port number `9900` with an integer of your choice between 9000 and 65535 (it needs to be unique from other users):

```
ssh -L 9900:localhost:9900 csimpson@polaris.alcf.anl.gov
module load conda
conda activate /grand/projects/WALSforAll/conda_environments/parsl
jupyter notebook --no-browser --port 9900
```

The shell will generate a path that looks like `http://localhost:9900/?token=xxxxx`. Copy and paste it in a local browser. Navigate to your copy of the workshop repository on the file system and open the notebook `0_molecular-design-with-parsl.ipynb`.

Instructions for OLCF

To open the tutorial notebook on Summit, you will need to create an ssh tunnel. From a shell on your computer, follow these instructions, but replace the port number `9900` with an integer of your choice between 9000 and 65535 (it needs to be unique from other users):

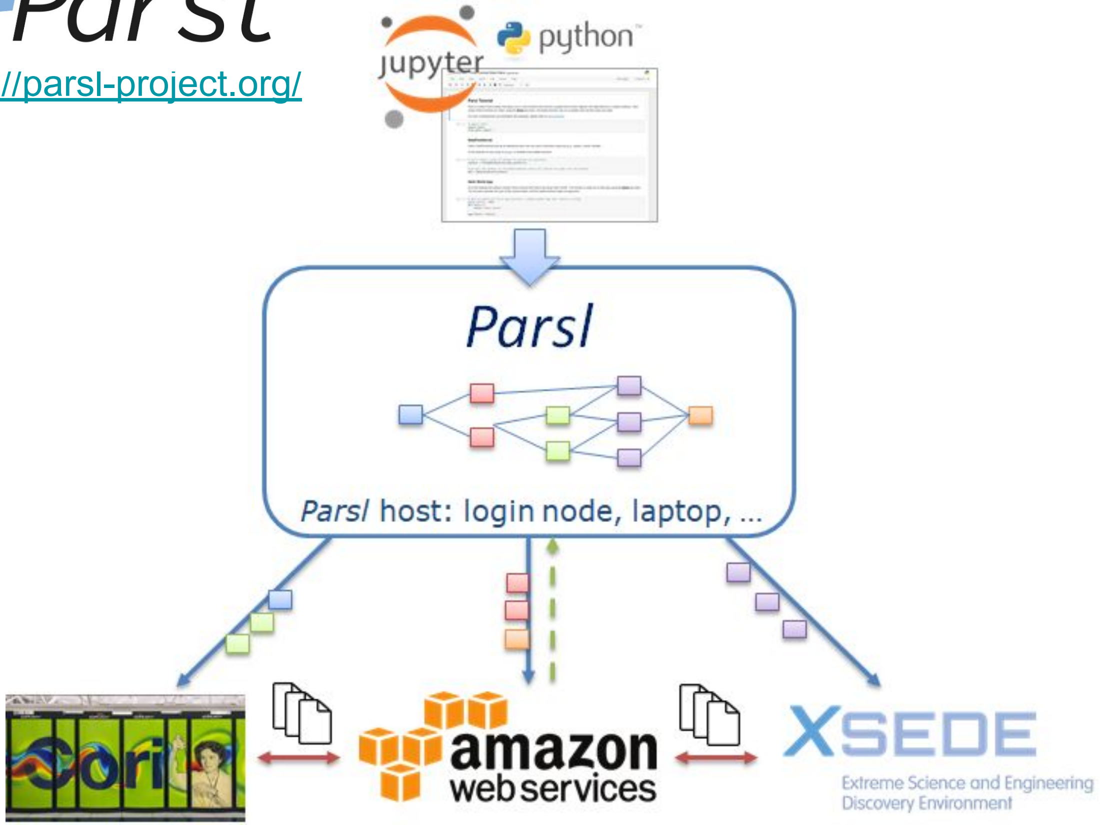
```
$ ssh -L 9900:localhost:9900 tskluzac@summit.olcf.ornl.gov

$ git clone https://github.com/CrossFacilityWorkflows/DOE-HPC-workflow-training.git

$ module load python

$ source activate /gpfs/alpine/world-shared/stf001/parsl_tutorial

$ jupyter notebook --no-browser --port 9900
```

The shell will generate a URL that looks like `http://localhost:9900/?token=xxxxx`. Copy and paste it in a local browser. Navigate to your copy of the workshop repository on the file system and open the notebook `mol-design-demo.ipynb`.

What is Parsl?

https://parsl-project.org/

- Started in 2017 by Y. Babuji, K. Chard, et al. at UChicago/ANL
- Parsl "provides an intuitive, pythonic way of parallelizing codes by annotating 'apps' "
- Apps execute concurrently while respecting data dependencies.
- "Write once, run anywhere" philosophy


jupyter
python
Parsl
Parsl host: login node, laptop, ...
Cori
amazon web services
XSEDE
Extreme Science and Engineering Discovery Environment

Motivation: The modern research computing landscape

Software is increasingly *assembled* rather than written

- High-level language to integrate components from many sources

Parallel and distributed computing is ubiquitous

- Increasing data sizes & plateauing sequential processing

Resources are increasingly heterogeneous (distributed)

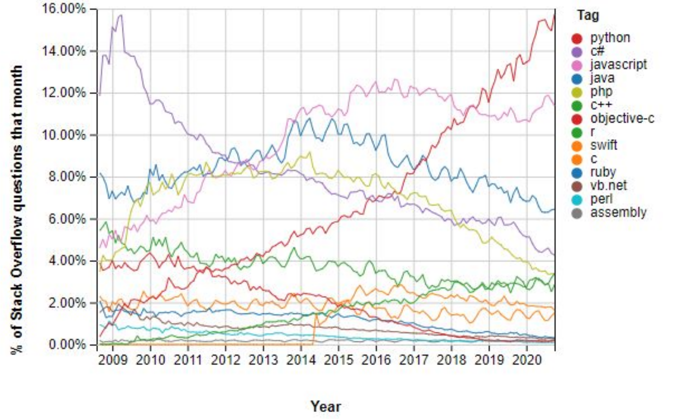
- Application components best run in different places

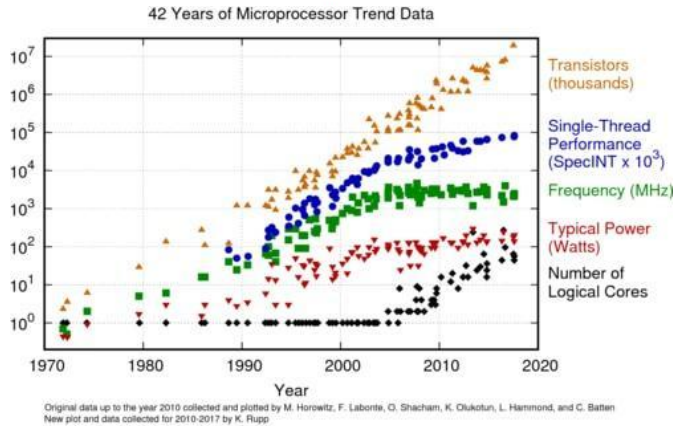
Python (& SciPy ecosystem) de facto standard language

- Libraries, tools, Jupyter, etc.

Parsl allows for the natural expression of parallelism in Python

funcX enables fire-and-forget remote and distributed execution of Python functions





Argonne NATIONAL LABORATORY Leadership Computing Facility

NERSC National Energy Research Scientific Computing Center

OAK RIDGE National Laboratory LEADERSHIP COMPUTING FACILITY

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

IP[y]:

🟩 Production
🟧 Prototype
🟥 Deprecated

Providers

Slurm	LSF	GridEngine	Kubernetes	AWS
PBS	Cobalt	HTCondor	Google	Ad hoc

Why Parsl?

- Easy to install, no database dependencies.
- Designed for large scale computing.
- Strong documentation and community support.
- Entirely Python orchestration: No bash scripts for workflows, no explicit slurm calls in the workflow.
- Not dependant on external infrastructure (i.e. like a DB)

- Most issues with running Parsl arise from its preferred setup of a master process trying to orchestrate resources with NERSC's busy slurm instance.
- How executor, provider and launcher parameters relate to slurm parameters is a learning process.
- Workers are stateless.

Parsl was made with HPC in mind.

- Parsl can schedule up to 1000-2000 apps/tasks per second.
- As a rule of thumb, in a high throughput case, the ideal duration of a task is larger than 0.01s times the number of nodes, i.e. for 100 nodes the duration of a task shall be longer than a second.
- Parsl has been tested to scale up to 250K workers on 8000 nodes. (Y. Babuji et al. https://doi.org/10.1145/3307681.3325400)
- For logging, Parsl will create one directory for each node, and 1 manager and N workers will write separate logs to this directory. Logging is pretty minimal at two lines per task unless debug logging is turned on.
- Bash apps communicate via files only. This may add to or suffer from file system congestions

Parsl executors scale to 2M tasks/256K workers

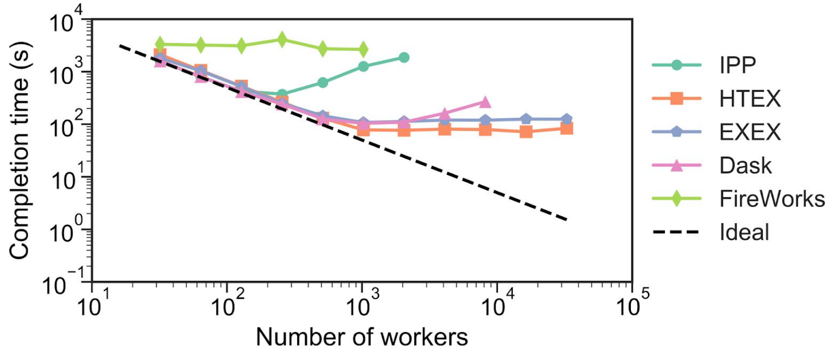
HTEX and EXEX outperform other Python-based approaches

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

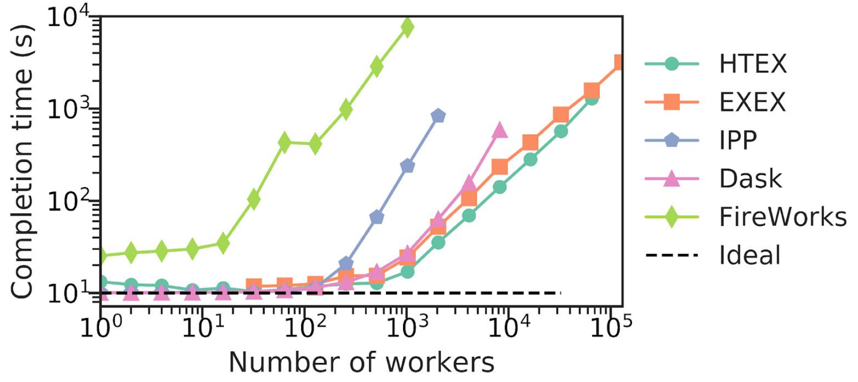
Framework	Maximum # of workers[†]	Maximum # of nodes[†]	Maximum tasks/second[‡]
Parsl-IPP	2048	64	330
Parsl-HTEX	65 536	2048*	1181
Parsl-EXEX	262 144	8192*	1176
FireWorks	1024	32	4
Dask distributed	4096	128	2617

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)



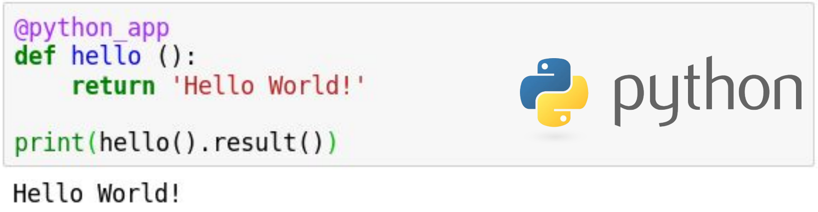
Parsl: parallel programming in Python

Apps define opportunities for parallelism

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

```
pip install parsl
```

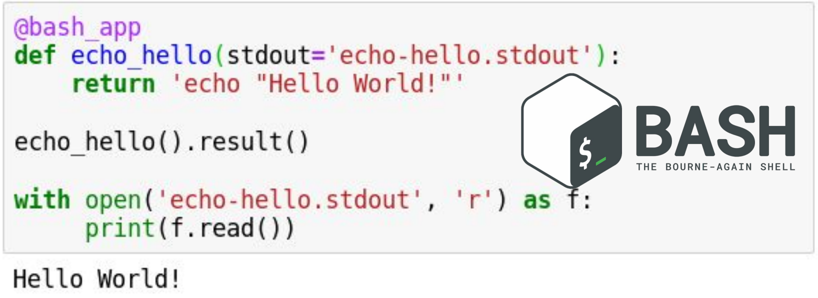
Apps return “futures”: a proxy for a result that might not yet be available



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

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

```
Hello World!
```

Apps run concurrently respecting dataflow dependencies. Natural parallel programming!



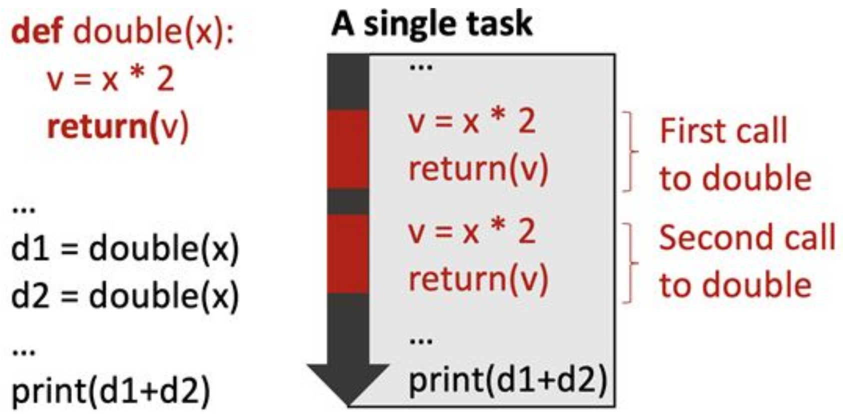
```
@bash_app
def echo_hello(stdout='echo-hello.stdout'):
    return 'echo "Hello World!"'

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://parsl-project.org/binder

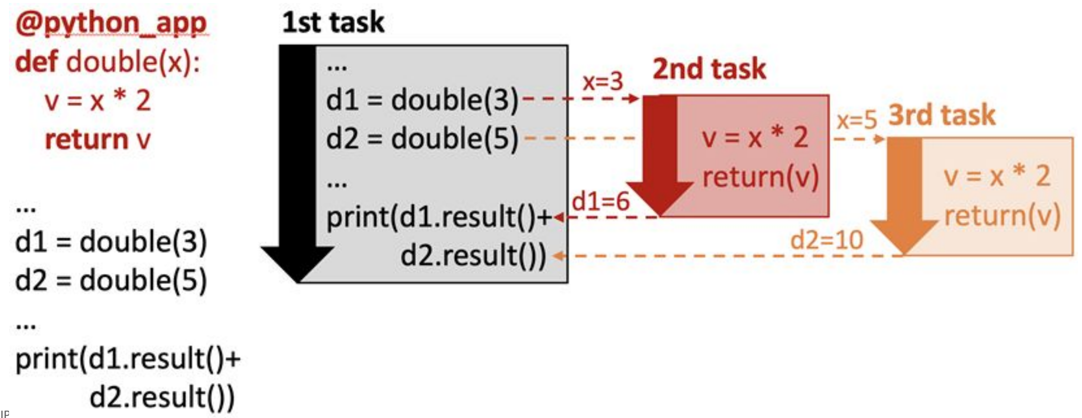
How does it work?



```
def double(x):
   v = x * 2
   return(v)

...
d1 = double(x)
d2 = double(x)
...
print(d1+d2)
```

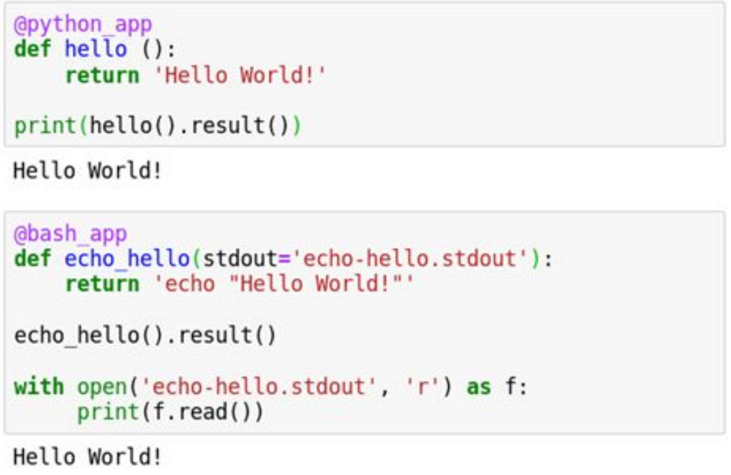
Python

Parsl



```
@python_app
def double(x):
   v = x * 2
   return v

...
d1 = double(3)
d2 = double(5)
...
print(d1.result()+
       d2.result())
```

How does it work?



```
@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!
```

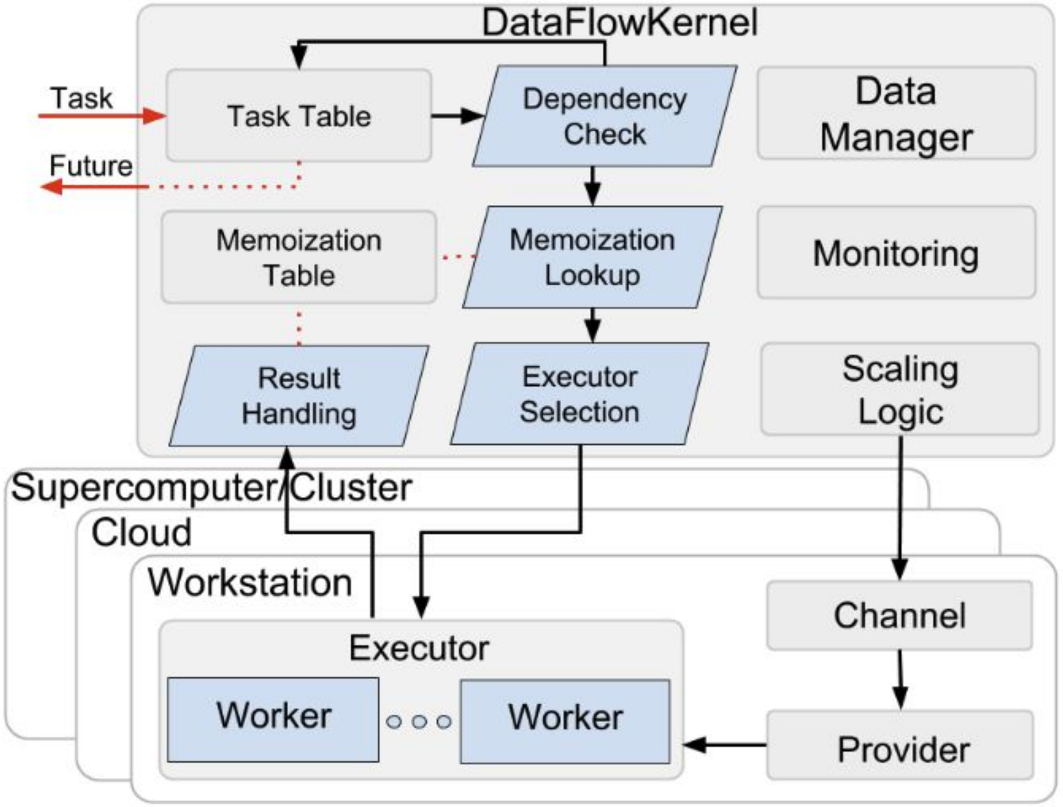
Developing a workflow is a two-step process:

- Annotate functions that can be executed in parallel as Parsl apps.
- Specify dependencies between functions using standard Python code.

- An App call returns immediately, generates a *task* in the DataFlowKernel (Parsl's task management engine) and a Future in the Python script.
- Futures can be passed to other apps as inputs, establishing a dependency.
- Dependencies are assembled implicitly into **directed acyclic graphs**
- Dependency graph is not computed in advance but dynamically built and updated while the Parsl script executes. It complete when the script finishes executing.

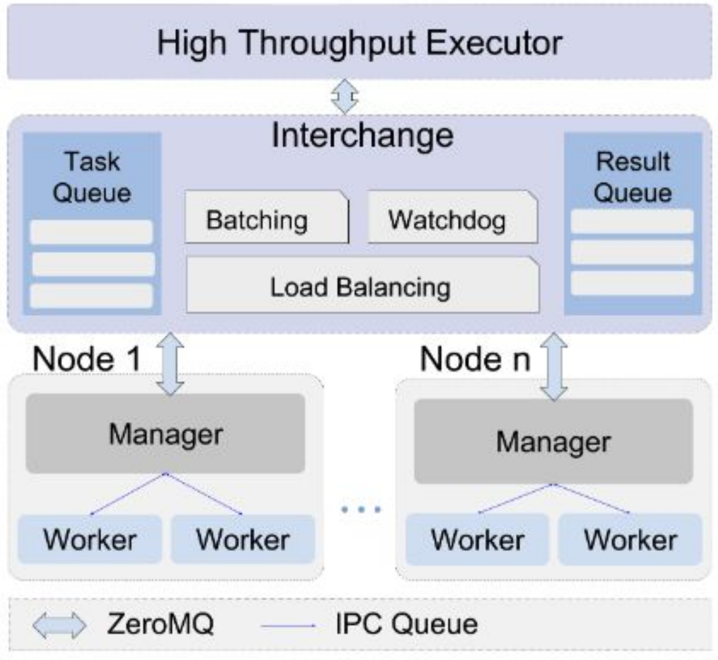
High Level Architecture

- ***DataFlowKernel (DFK)*** is responsible for constructing and orchestrating the execution of the task graph.
- Uses one or more connected ***Executors***.
- ***Tasks*** are dispatched to Executor(s) which run in blocks of nodes from Providers.
- *Tasks* are executed concurrently if their (data-)dependencies are met
- *Tasks* will run in parallel if the Executor has enough resources/workers.



High Throughput Executor (HTEX)

- Engineered to support up to 2000 nodes, millions of sub-second tasks, and multi-day workflow campaigns with high fault tolerance.
- Manager is a multi-threaded agent responsible for a single node, initializing workers based on HTEX configuration (i.e., *workers_per_node*).
- Task & results are prefetched and batched to minimize communication overhead.
- Managers and the interchange exchange periodic heartbeat messages.



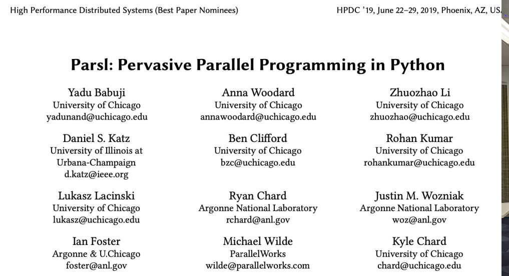
(a) HTEX: High Throughput Executor

Parsl++

You

I'm interested in learning more about Parsl!

I'm becoming proficient in Parsl!



High Performance Distributed Systems (Best Paper Nominees) HPDC '19, June 22–29, 2019, Phoenix, AZ, US

Parsl: Pervasive Parallel Programming in Python

Yadu Babuji University of Chicago yadunand@uchicago.edu	Anna Woodard University of Chicago annawoodard@uchicago.edu	Zhuozhao Li University of Chicago zhuozhao@uchicago.edu
Daniel S. Katz University of Illinois at Urbana-Champaign d.katz@ieee.org	Ben Clifford University of Chicago bzc@uchicago.edu	Rohan Kumar University of Chicago rohankumar@uchicago.edu
Lukasz Lacinski University of Chicago lukasz@uchicago.edu	Ryan Chard Argonne National Laboratory rchard@anl.gov	Justin M. Wozniak Argonne National Laboratory woz@anl.gov
Ian Foster Argonne & U.Chicago foster@anl.gov	Michael Wilde ParallelWorks wilde@parallelworks.com	Kyle Chard University of Chicago chard@uchicago.edu

Read the Parsl paper! [HPDC '19]

Attend Parsl Fest

funcX

Federated function as a service

Check out funcX (FaaS system built atop Parsl)

Launching Parsl Project

Join the Parsl community on Slack!

Argonne National Laboratory Leadership Computing Facility

NERSC National Energy Research Scientific Computing Center

Oak Ridge National Laboratory Leadership Computing Facility

Parsl is commonly installed via conda

```
$ module load python
$ conda create --name parsl
$ source activate parsl
(parsl) $
```

Install parsl or with pip or through conda forge:

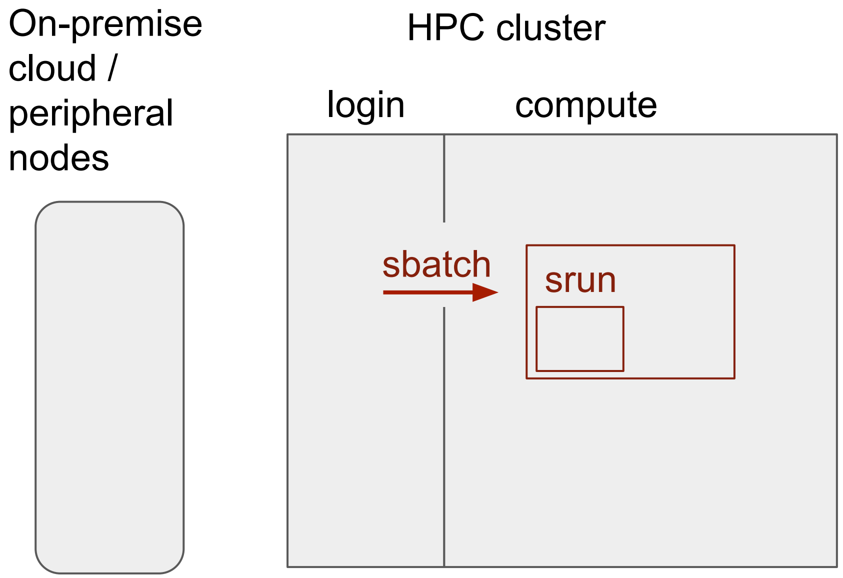
```
(parsl) $ python -m pip install parsl
```

or

```
(parsl) $ conda install -c conda-forge parsl
```

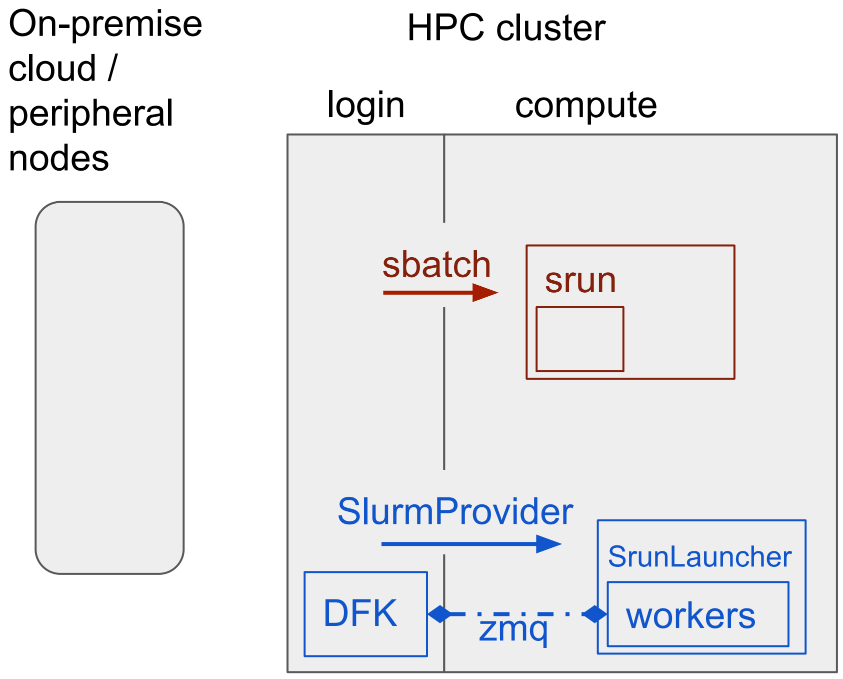
Configuring Parsl for your HPC system

- For Parsl to work as expected it is crucial to set up the config properly.
- The DFK will run in the process where the config is loaded, i.e. the main script



(Semi)permanent "Pilot" process hosts the DFK

- For long or semipermanent workflows that may react on external commands/data.
- DFK process placed on login nodes (or workflow nodes) or in SCRONTAB/workflow qos (NERSC).
- Must keep DFK process alive after logout.
- Parsl needs to use LRM provider (e.g. SlurmProvider) to get resources for the workers.



SlurmProvider = sbatch

(Semi)permanent "Pilot" process hosts the DFK

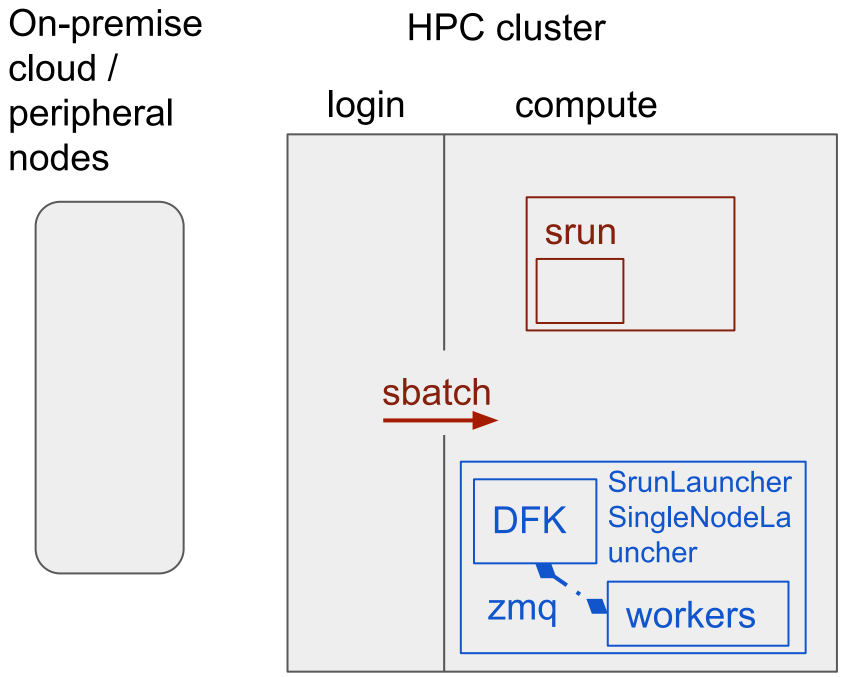
```
import parsl
from parsl.config import Config
from parsl.providers import SlurmProvider
from parsl.launchers import SrunLauncher
from parsl.executors import HighThroughputExecutor

config = Config(
     executors=[
     HighThroughputExecutor(
          label='PM_HTEX_multinode',
          cores_per_worker=2,
          provider=SlurmProvider(
                    #'regular',  # Partition / QOS (NERSC doesn't use partitions though)
                    nodes_per_block=2,
                    scheduler_options=' #SBATCH -C cpu',
                    worker_init='module load python; source activate parsl',
                    launcher=SrunLauncher(overrides='-c 128 -q regular'),
                    walltime='00:10:00',
                    # Slurm scheduler on Cori can be slow at times, increase the command timeouts
                    cmd_timeout=120, # Increase timeout for unresponsive schedulers
          ),
     )]
)
parsl.load(config)
```

https://docs.nersc.gov/jobs/workflow/parsl/#example-creating-a-parsl-script-for-a-high-throughput-scenario

Argonne National Laboratory Leadership Computing Facility · NERSC National Energy Research Scientific Computing Center · Oak Ridge National Laboratory Leadership Computing Facility

Headless config with DFK inside allocation

- For "boxed" workflows or Jupyter kernels on compute nodes.
- DFK process runs in head node of preallocated block of resources.
- Can run unattended.
- No need to use a LRM provider class, only needs LRM launcher for multi node access.


On-premise cloud / peripheral nodes
HPC cluster
login
compute
srun
sbatch
DFK
SrunLauncher SingleNodeLauncher
zmq
workers

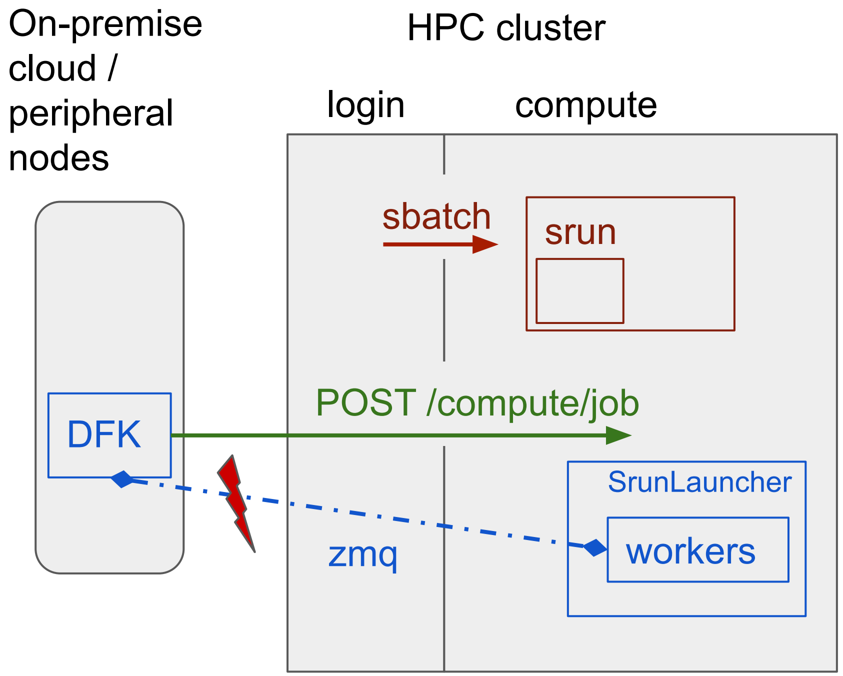
Headless config with DFK inside allocation

```
import parsl
from parsl.config import Config
from parsl.providers import LocalProvider
from parsl.launchers import SrunLauncher
from parsl.channels import LocalChannel
from parsl.executors import HighThroughputExecutor

config = Config(
      executors=[HighThroughputExecutor(
            label='PM_HTEX_headless',
            max_workers=1,  # one worker per manager / node
            provider=LocalProvider(
                  channel=LocalChannel(script_dir='.'),
                  nodes_per_block=2,
                  launcher=SrunLauncher(overrides='-c 32'),
                  cmd_timeout=120,
                  init_blocks=1,
                  max_blocks=1
            ),
      )],
      strategy=None,
)
parsl.load(config)
```

https://docs.nersc.gov/jobs/workflow/parsl/#example-headless-workflow-launched-from-node-0

Why not put the DFK on cloud/peripheral nodes?

- For permanent workflows with a web frontend.
- DFK process runs in on on-premise Cloud (i.e. Spin at NERSC).
- If LRM isn't exposed, it needs a new provider that wraps facility APIs for job execution.
- Can be problematic with all the ports that the zmq communication needs.

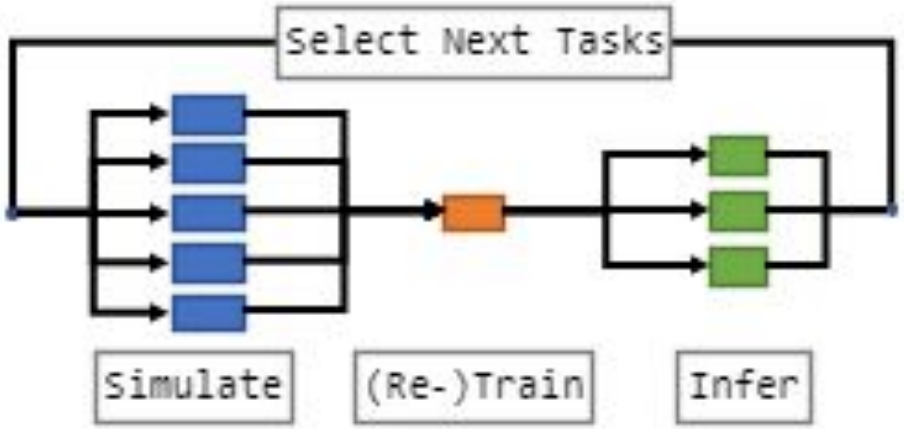


Tutorial example: ML-in-the-loop materials design

Aim: identify high value molecules (high ionization energy) among a search space of billions of candidates

Problem: Simulation is expensive

Solution: Create an active learning loop that couples simulation with machine learning to simulate only high value candidates



https://github.com/ExaWorks/molecular-design-parsl-demo

If you can't get the notebook up at NERSC | ALCF | OLCF, go there and use binder.