Goals

- Easy to write Python workflows that glue together external programs and Python functions
- Easy to run in parallel on diverse resources
- Easy to install: `pip install parsl`
- Open source (Apache 2.0 license)
- Open community

Scientific Workflows

Scientific Workflows are classified into different categories based on their use cases:

- **HPC/HTC workflow**
  - O(10) proteins
  - O(100k) drug candidates
  - 1M docking tasks, hundreds of MD models to find candidates for experiments

- **Machine learning workflow**
  - O(Ms) of data used to train a model
  - Model applied to O(Ms) of possible materials

- **Interactive workflow**
  - O(1M) cosmic ray events
  - Iterative collection, curation, analysis, visualization

### Write once, run anywhere

On clouds, clusters, supercomputers

Paral scripts are independent of the execution environment. A single script can be executed on one or more execution resources without modifying the script.

### Implicit dataflow

Apps execute concurrently while respecting data dependencies

Paral creates a dynamic graph of tasks and their data dependencies. Tasks are only executed when their dependencies are met.

### Scalable Jupyter notebooks

Easily manage execution across distributed resources

Paral works seamlessly with Jupyter notebooks allowing apps within a notebook to be executed in parallel and on remote resources.

### Automated data movement

Implicit wide area staging

Paral handles the complexity of ensuring data is in the right place at the right time for computation.

### Execution management

Handles failures and elasticity

Paral uses checkpointing and automatic retries as a resilience mechanism to handle failures. Paral apps can be containers in resource pools that grow and shrink elasticity as needed.

### Configuration: Use arbitrary resource(s)

```
Comet_config = Config(
    executors=[
        IPyParallelExecutor(
            label='comet_ipa_multinode',
            provider=SlurmProvider(
                'compute'))
    ])
parsl.load(Comet_config)
```

### App definition: Run Python and bash apps

```
@bash_app
def generate(outputs=[]):
    # return a random number from 1 to 10
    return '"echo \$((( RANDOM % 10 ) + 1 )) &> {outputs[0]}"'

@python_app
def total(inputs=[]):
    total = 0
    for i in inputs:
        with open(i, 'r') as f:
            total += sum([int(line) for line in f])
    return total
```

### Execution: Transparent parallelization based on data dependencies

```
# Create 5 files with random #s
output_files = []
for i in range (5):
    output_files.append(generate(outputs=['r%s.txt' % i]))

# Calculate the sum of the random numbers
r = total(inputs=[i.outputs[0] for i in output_files])
print (r.result())
```