Parsl: Productive Parallel Programming in Python

http://parsl-project.org

Yadu Babuji[*], Anna Woodard[*], Zhuozhao Li[*], Daniel S. Katz[°], Ben Clifford[*], Ryan Chard[+], Justin M. Wozniak[*], Ian Foster[⚑], Michael Wilde[*⚐], Kyle Chard[*⚑]*

[*]University of Chicago, [+]Argonne National Laboratory; ⚑Globus; ⚐Parallel Works
[°]National Center for Supercomputing Applications, University of Illinois at Urbana-Champaign

Motivation

- Software is increasingly assembled rather than written, using high-level languages
- New scientific methods demand many-task, parallel & distributed computing
- Python is pervasive in science and engineering
- Developers require parallel runtimes & remote execution for new science applications

Parsl aims to bring together the simplicity and high productivity possible with Python with the complex workflow patterns and extreme scale demanded by scientific computing

Why Parsl?

- Pure Python: Easily parallelize Python code
- Implicit dataflow: Apps execute concurrently while respecting data dependencies
- Write once, run anywhere: On clouds, clusters, and supercomputers
- Automated data management: Implicit and flexible wide area and local staging
- Toolkit approach: Different executors optimized for different classes of applications
- Open source & open community: Guided by users, with some executors and other components from outside the core team

Programming with Parsl

```
@python_app
def pi(num_points):
    from random import random
    inside = 0
    for i in range(num_points):
        x, y = random(), random()  # Drop a random point in the box.
        if x**2 + y**2 < 1:        # Count points within the circle.
            inside += 1
    return (inside*4 / num_points)


# App that computes the mean of three values
@python_app
def mean(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 mean of the three estimates
mean_pi  = mean(a, b, c)

# Print the results
print("Average: {:.5f}".format(mean_pi.result()))
```

@python_app is a Python decorator that introduces asynchronous behavior to the functions

futures are a proxy for an asynchronous computation. When decorated functions are invoked, they return **futures**

Passing **futures** from one app to another introduces a dependency in the task graph

Wait for **future**

Features

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

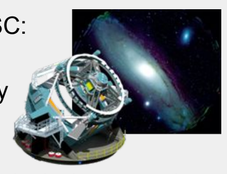
Reproducibility. Capture of workflow provenance in the task graph

Applications (users performing data analysis, simulation, etc.)

ArcticDEM: Satellite Image Processing

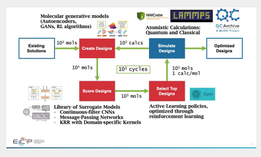
Months-to-years of robustly processing massive amounts of data for sharing

LSST-DESC: Simulated Sky Survey



Running containerized apps on entire HPC systems (Cori, Theta) for days to create simulated LSST images

Designing new battery materials with reinforcement learning



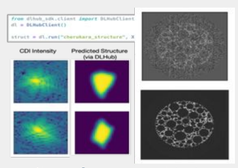
High-throughput ML on heterogeneous systems, combining ML training, simulation, model selection, and inference

Platforms (tools on which users analyze data, simulate, etc.)

QCArchive: A MolSSI Project

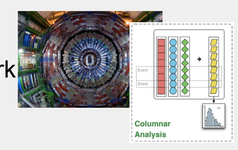
Compile, aggregate, query, and share quantum chemistry data on diverse systems

DLHub: Machine Learning Inference



Interactive execution of user-provided machine learning models in real-time

Coffea: Column Object Framework for Effective Analysis



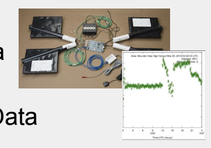
Back-end-agnostic data processing libraries for granular event-based High Energy Physics analysis

Education (teaching and embedding parallelism)

Lewis & Clark College's Campus Cluster Interface

Default method for submitting tasks to the campus cluster, hiding the HPC scheduler

QuarkNet: Teaching Data Science with Cosmic Ray Data



Interactive notebooks that enable high school students to learn data science at scale

Configuration

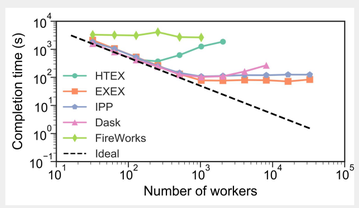
```
Comet_config = Config(
    executors=[
        HighThroughputExecutor(
            label='comet_htex_multinode',
            provider=SlurmProvider(
                'compute'
        ))])
parsl.load(Comet_config)
```

Use arbitrary resources

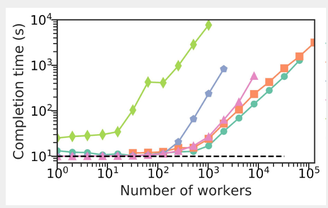
- Providers for clouds, clusters, supercomputers
- Separation of code and config
- Write once, run anywhere

- Configuration
  - How to Configure
  - Comet (SDSC)
  - Cori (NERSC)
  - Stampede2 (TACC)
  - Frontera (TACC)
  - Theta (ALCF)
  - Cooley (ALCF)
  - Blue Waters (Cray)
  - Summit (ORNL)
  - CC-IN2P3
  - Midway (RCC, UChicago)
  - Open Science Grid
  - Amazon Web Services
  - Kubernetes Clusters
  - Ad-Hoc Clusters

Scaling Performance



Strong scaling (50k 1-second tasks)



Weak scaling (10 1-second tasks per worker)

- Scaling data on Blue Waters
- Outperforms other Python-based approaches
- Scales beyond ~2M tasks
- Easily scales to >2K nodes, with >1K tasks/s


ParslFest 2019 attendees

- Install from PyPI or Conda Forge
- Open source (Apache 2.0 license)
- Open community (~200 GitHub stars, ~35 contributors, used by ~30 projects)
- http://parsl-project.org

Parsl is supported by the National Science Foundation under awards 1550476, 1550475, 1550528, 1550562, 1550588 and Argonne National Laboratory's Laboratory Directed Research and Development (LDRD) program