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
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](http://swift-lang.org)
Very fast, highly portable, pervasively parallel dataflow
Orchestrates apps passing files

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

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!

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:

```python
@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:

```python
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
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
Separation of code and execution

```python
# sample_configs.py

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

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).
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
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

<table>
<thead>
<tr>
<th>Framework</th>
<th>Maximum # of workers</th>
<th>Maximum # of nodes</th>
<th>Maximum tasks/second</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parsl-IPP</td>
<td>2048</td>
<td>64</td>
<td>330</td>
</tr>
<tr>
<td>Parsl-HTEX</td>
<td>65 536</td>
<td>2048</td>
<td>1181</td>
</tr>
<tr>
<td>Parsl-EXEX</td>
<td>262 144</td>
<td>8192</td>
<td>1176</td>
</tr>
<tr>
<td>FireWorks</td>
<td>1024</td>
<td>32</td>
<td>4</td>
</tr>
<tr>
<td>Dask distributed</td>
<td>4096</td>
<td>128</td>
<td>2617</td>
</tr>
</tbody>
</table>

Monitoring and visualization

Workflows

<table>
<thead>
<tr>
<th>Name</th>
<th>Version</th>
<th>Owner</th>
<th>Status</th>
<th>Runtime (s)</th>
<th>Tasks</th>
<th>Actions</th>
</tr>
</thead>
<tbody>
<tr>
<td>test_udp_simple.py</td>
<td>2019-02-20 22:16:43.570004</td>
<td>zhuozhao</td>
<td>Completed</td>
<td>25.218577</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>test_fan_in_out.py</td>
<td>2019-02-20 22:27:05.407903</td>
<td>zhuozhao</td>
<td>Completed</td>
<td>151.513495</td>
<td>12</td>
<td></td>
</tr>
</tbody>
</table>

Workflow view

Task view
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)
Parsl is an open-source Python community

Parsl - Parallel Scripting Library  http://parsl-project.org

Manage topics

3,328 commits  97 branches  0 packages  33 releases  1 environment  35 contributors  Apache-2.0
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