Parsl: A Parallel Programming Library for Python

Kyle Chard (chard@uchicago.edu)
On behalf of the Parsl community (including Yadu Babuji, Ben Clifford, Ian Foster, Dan Katz, Zhuozhao Li, Mike Wilde, Anna Woodard)

http://parsl-project.org
Example workload: simulating images from the Vera C. Rubin Observatory

- **Catalog 1**: 189 sensors x ~10,000s of instance catalogs
- **Catalog 2**
- **Catalog 3**

**Node-sized bundles**

- **Bundler**
- **Executor**

**JSON description**

**Executor**

- **256K cores for 3 days**
- **128K cores for 3.5 days**

**Executor**

- **x 4000 nodes**
- **x 189 x ~10,000s**

**Example workload**: simulating images from the Vera C. Rubin Observatory
Example workload: Applying extreme-scale AI to screen billions of druglike molecules against COVID-19 proteins
Distribution, parallelism, and composition

Parallel and distributed computing is ubiquitous
  – Increasing data sizes combined with plateauing sequential processing power

Software is increasingly assembled rather than written
  – High-level language to integrate and wrap components from many sources

Python (and the SciPy ecosystem) has established itself as one of the most productive and popular environments for research
  – Thriving ecosystem of libraries, tools, Jupyter, etc.
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 dataflow dependencies. Natural parallel programming!

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

pip install parsl

Try Parsl: https://parsl-project.org/binder
Parsl’s dataflow programming model delivers intuitive parallel programming

- Programming paradigm in which program is assembled as a directed graph of data flowing between tasks
- Intuitive way to think about parallelism (tasks run independently when data slice ready)
- Parsl’s dataflow model allows data to be passed between Apps
  - Python types and objects
  - Files (local or via HTTP, FTP, or Globus)
Data-driven example: parallel geospatial analysis

Land-use Image processing pipeline for the MODIS remote sensor
Expressing parallelism using Parsl

1) Wrap the science applications as Parsl Apps:

```python
@bash_app
def landuse(img, outputs=[]):
    return './landuse_sim.sh {} {}' .format(img, outputs[0])

@python_app
def colorize(img, num_chunks):
    return color_package(img, num_chunks)

@python_app
def analyze(land_chunks, color_chunks):
    return combine(land_chunks, color_chunks)
```
Expressing a many task workflow in Parsl

2) Execute the parallel workflow by calling Apps:

```python
lchunks = []

for i in range (nchunks):
    lchunks.append(landuse(img, outputs=[File('l%s.txt' % i)]))

colored = colorize(img, num_chunks=5)

results = analyze(lchunks, colored)
```
Decomposing parallelism into a dynamic task-dependency graph for distributed execution
Enabling portable Parsl programs: providers

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

Growing support for various schedulers and cloud vendors
Separation of code and execution

Choose execution environment at runtime. Parsl will direct tasks to the configured execution environment(s).
Parsl implements a Python’s Concurrent.futures executor (runtime) 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, RADICAL-Cybertools, Flux
Parsl executors scale to 256K concurrent workers

HTEX and EXEX outperform other Python-based approaches

Parsl scales to more than 250K workers (8K nodes) and ~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>

Strong scaling (50K 1s tasks)

Weak scaling (10 1s tasks per worker)

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

https://parsl-project.org/parslfest.html
Parsl is an open-source Python community (parsl-project.org)
funcX: Parsl as a service for remote computing

Common Parsl use case: I want to run my computation on one or more remote clusters, clouds, supercomputers from my PC

Cloud-hosted managed compute service built on Parsl
FuncX: Fire-and-forget remote function execution

**funcX Service:**
- Single reliable cloud service (REST)
- Register, share, run functions
- Fire-and-forget execution: funcX will manage execution, store results in the cloud, handle errors, etc.

**Endpoints:**
- User-deployed and managed
- Dynamically provision resources, deploy containers, execute functions, catch exceptions, etc.
- Exploit local architecture/accelerators

**Choose where to execute functions**
- Closest, cheapest, fastest, accelerators …
Parsl provides productive, safe, scalable, and flexible parallelism in Python

Productive: 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 and portable: efficient execution from laptops to the largest supercomputers

Flexible: programs composed from existing components and then applied to different resources/workloads
ExaWorks: ECP Workflows Project

- Community-based project to support workflows in ECP
  - Users, workflow developers, facilities, vendors
- Technical development
  - ExaWorks SDK: packaged and compatible workflows components
  - PSI/J: Asynchronous Python library for scheduler portability
- Community development
  - Organizing a series of summits to bring the workflows community together
    - [https://exaworks.org/summit.html](https://exaworks.org/summit.html)
  - Contributing to the Workflows Community Initiative ([https://workflows.community](https://workflows.community))

[https://exaworks.org](https://exaworks.org)
Questions?

parsl-project.org

parsl-project.org/binder