Scalable Parallel Programming in Python with Parsl

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

Ian Foster
Argonne & U.Chicago
foster@anl.gov

Michael Wilde
ParallelWorks
wilde@parallelworks.com

Kyle Chard
University of Chicago
chard@uchicago.edu

ABSTRACT

Python is increasingly the *lingua franca* of scientific computing. It is used as a higher level language to wrap lower-level libraries and to compose scripts from various independent components. However, scaling and moving Python programs from laptops to supercomputers remains a challenge. Here we present Parsl, a parallel scripting library for Python. Parsl makes it straightforward for developers to implement parallelism in Python by annotating functions that can be executed asynchronously and in parallel, and to scale analyses from a laptop to thousands of nodes on a supercomputer or distributed system. We examine how Parsl is implemented, focusing on syntax and usage. We describe two scientific use cases in which Parsl's intuitive and scalable parallelism is used.

ACM Reference Format:
Yadu Babuji, Anna Woodard, Zhuozhao Li, Daniel S. Katz, Ben Clifford, Ian Foster, Michael Wilde, and Kyle Chard. 2019. Scalable Parallel Programming in Python with Parsl. In *Practice and Experience in Advanced Research Computing (PEARC '19), July 28-August 1, 2019, Chicago, IL, USA.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3332186.3332231

1 INTRODUCTION

Python is widely used in science and engineering. It is a high-level language, is easy to learn and intuitive, and does not require a compiler. However, for use in science, it has several challenges, most notably related to parallelization and scalability. Parsl [2, 3], an open-source parallel programming library for Python, aims to address these needs by providing simple, reliable, scalable, and flexible parallelism. Parallel programs are constructed in Python using a compositional model, in which components written in Python and other languages (e.g., binaries, shell commands, or external applications) can be easily used in combination. Parsl allows data-oriented workflows to be constructed by implicitly linking together invocations of these components via shared data in the form of Python objects or files.

Parsl is designed to facilitate parallel execution at a wide range of scales. Run time of individual tasks can range from seconds to hours, and programs can be scaled transparently to work on platforms from a laptop to a supercomputer or distributed computing system. It is also designed to be accessible from within Jupyter notebooks, thus enabling users to control large-scale execution from the user-centric notebook environment. To enhance usability, Parsl stays as close to standard Python as possible, and maintains compatibility with the growing SciPy ecosystem.

Parsl includes two new constructs. The first is an *App*, which is indicated by a decorator around a Python function that can be executed asynchronously, and that declares other information such as input and output dependencies for linking together with other Apps. This provides an intuitive, implicit, and natural way of expressing parallelism. When an App is called, Parsl immediately returns the second new construct, an object called a future, which is a placeholder for a return value from an App that may not have completed.

Parsl takes a very different approach to parallelism than prior efforts that rely on domain specific languages (DSL) [13, 17], configuration-based models [8, 12, 14], graph descriptions [15, 16], and compiled language extensions [5] to support such composition and parallelism. When compared with other Python libraries designed to support parallelism (e.g., Dask [10] and Fireworks [15]), Parsl can scale to orders of magnitude larger computing environments and supports a much broader range of parallel use cases (e.g., low latency, wide area data management, execution on clouds, clusters, and supercomputers).

In this paper we describe how parallel programs can be constructed in Parsl, how Parsl is designed to support different scalability and parallelism requirements, and how Parsl has been used in science.

2 DESIGN AND IMPLEMENTATION

Parsl is designed for Python and implemented as a Python library. When executing a Parsl program, Parsl intercepts invocations of Apps and returns futures. A future is an object that can be used to access the results of an asynchronous computation. Parsl stores

ACM ISBN 978-1-4503-7227-5/19/07…$15.00
https://doi.org/10.1145/3332186.3332231

invocations of Apps in a dynamic task graph. The graph is dynamic in the sense that it is not defined a priori and instead is built during execution. The task graph is a directed acyclic graph (DAG), where nodes represent tasks and edges represent input/output data passed between tasks.

Parsl's runtime system is responsible for managing the execution of the task graph on configured resources. The runtime converts the App invocation (including the function body and any input arguments) into a serialized form that can run locally or be transferred to and executed on a remote system. The runtime also manages execution by updating the DAG when new tasks are created and queuing serialized tasks for execution when it determines dependencies are met. During execution, Parsl monitors the state of all tasks, captures errors, and implements retry logic, if configured. When a task completes executing, the result (or error) is serialized and passed back to the Parsl program via a future. Parsl's runtime is entirely event-driven. Asynchronous callbacks on dependent futures communicate results to the Parsl program, trigger DAG updates, and result in queuing of dependent tasks.

Parsl is designed to function efficiently in a range of scenarios, from a few very short duration tasks to millions of long-running tasks scattered across hundreds of thousands of worker nodes running on a variety of execution resources including clouds and clusters. Consequently, we have designed Parsl around a modular execution architecture with two primary components, providers and executors. Providers abstract execution resources while executors abstract the mechanism by which the tasks are executed (e.g., threads, pilot jobs, distributed execution). The provider interface is a simple Python-based abstraction that supports three actions: submit a job for execution, query the status of a job, and cancel a job. Parsl implements providers for local execution (fork), Amazon and Google cloud platforms (using cloud-specific APIs), and for cluster and supercomputer schedulers such as Slurm, Torque/PBS, HTCondor, and Cobalt.

To support a broad variety of use cases, Parsl includes four core executors. The Thread Pool Executor is designed for local, thread-based execution of lightweight tasks. The Low Latency Executor (LLEX) is designed for small-scale deployments (tens of nodes) and provides very fast task execution by utilizing a bare bones executor implementation and persistent connections between Parsl and configured workers. The High Throughput Executor (HTEX) implements a pilot job model that scales to 2,000 nodes and millions of tasks, deploying a multi-threaded Python-based manager to each node and using asynchronous messaging to manage execution of tasks. Finally, the Extreme Scale Executor (EXEX) is designed for deployments with thousands of nodes and longer running tasks; it deploys a distributed MPI job across a large-scale computer network and relies on MPI communication to coordinate execution of tasks.

Parsl simplifies program portability by separating code and configuration. A Parsl program is executed by loading a Python-based configuration object that defines the providers and executors to be used as well as user-specific options describing how these resources should be used. The configuration object can be be shared and customized, both manually and programmatically, for different deployments. An example of the simplest possible configuration object, corresponding to a thread pool running locally, is shown in Listing 1. An example configuration object for Stampede2 at the Texas Advanced Computing Center is shown in Listing 2. This configuration uses HTEX to submit tasks from a login node (LocalChannel). It requests an allocation of 256 nodes, from the `normal` partition, for up to six hours.

```python
from parsl.config import Config
from parsl.executors import ThreadPoolExecutor

thread_config = Config(
    executors=[ThreadPoolExecutor(max_threads=2)]
)
```

Listing 1: Example of a Parsl configuration object for running lightweight tasks in a thread pool on the local machine.

```python
from parsl.addresses import address_by_hostname
from parsl.channels import LocalChannel
from parsl.config import Config
from parsl.data_provider.scheme import GlobusScheme
from parsl.executors import HighThroughputExecutor
from parsl.providers import SlurmProvider

uuid = "ceea5ca0-89a9-11e7-a97f-22000a92523b"

stampede_config = Config(
    executors=[
        HighThroughputExecutor(
            label="stampede2",
            worker_debug=False,
            address=address_by_hostname(),
            provider=SlurmProvider(
                channel=LocalChannel(),
                nodes_per_block=256,
                partition='normal',
                worker_init='source activate parsl',
                walltime="6:00:00"
            ),
            storage_access=[
                GlobusScheme(
                  endpoint_uuid=uuid,
                  endpoint_path="/",
                  local_path="/"
                )
            ]
        )
    ],
)
```

Listing 2: Example of a Parsl configuration object for Stampede2.

Parsl also provides a range of other features that are needed by scientific applications, including wide area data management using Globus [6], HTTP, and FTP; fault tolerance with workflow-level checkpointing and App caching; automated elasticity on clouds and clusters; multi-site execution across computational resources;

resource matching to map Apps to appropriate hardware; workflow and fine grain resource monitoring; and management of container-based applications.

3 PARALLEL PROGRAMMING WITH PARSL

We will now illustrate, with examples, how to use some key elements of programming with Parsl. Parallel programming is implemented in Parsl with Apps. A Python function can be converted into a Parsl App simply by prepending the appropriate decorator. For an App that executes Python code in parallel, the `@python_app` decorator is used. If the Python function instead returns a string of Bash code to be executed in parallel, it is designated with the `@bash_app` decorator. An App can be called from standard Python code, while Parsl handles both asynchronous and parallel execution. The following is a basic example of a Python App:

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

When a Parsl App is invoked, the Parsl runtime registers an asynchronous task, and a future object is immediately returned. Parsl futures implement an asynchronous, non-blocking method `future.done()` to monitor the App execution status. This method returns true if the App has finished, and otherwise returns false. A synchronous method `future.result()`, is also provided, which will block until the computation has completed and then return a value to the calling program. For Python Apps, the `future.result()` method yields the return value of the wrapped Python function, while for Bash Apps, the value that is returned is the UNIX exit code. If either type of App fails to complete, an exception will be raised.

Parsl apps are called using the standard Python syntax. For example,

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

invokes the `hello` App defined above, which returns a future object. Calling the `result` method of the future returns the string "Hello, World!"

Input data are passed as arguments to the App invocation. Output data are passed back either as a return value or an output file. Three Python types are supported as either inputs or outputs: 1) arbitrary, serializable Python objects, 2) Parsl `File` objects (described in more detail below), and 3) Parsl futures. To examine how this works, consider the following App:

```
@python_app
def product(a, b):
    return a * b
```

Calling `product` with the arguments 2 and 3 will return a future; calling the `result()` method of that future will yield 6. Similarly, one can pass futures as arguments. Parsl will wait until the input task has completed, then dispatch the dependent task with the input futures replaced by their results. This is illustrated in the example below, which prints 120.

```
f1 = product(2, 3)
f2 = product(4, 5)

print(product(f1, f2).result())
```

```
import parsl
from parsl.app import python_app

parsl.load()

@python_app
def divide(numerator, denominator):
    return numerator / denominator

# Produce a divide by zero exception
future = divide(10, 0)

# Catch and handle the exception
try:
     future.result()
except ZeroDivisionError:
     print('Oops! You tried to divide by 0!')
```

Listing 3: An example of exception handling with Parsl.

The special keyword arguments `inputs` and `outputs` are provided to allow developers to dynamically specify collections of inputs or outputs. For example,

```
@python_app
def product(inputs=[]):
    result = 1
    for i in inputs:
        result *= i
    return result

f1 = product([2, 3])
product([f1, 4, 5]).result()
```

will also print 120.

As Parsl apps are potentially executed remotely, they must contain all required dependencies in the function body. An example is shown below that makes use of `time` module to wait five seconds before returning.

```
@python_app
def slow_hello():
    import time
    time.sleep(5)
    return 'Hello World!'
```

Parsl allows the developer to handle exceptions and errors within an App without halting the program and potentially interrupting complex workflows. A minimal example is shown in Listing 3. In addition to being able to capture exceptions raised by a specific App, Parsl also raises dependency errors when Apps are unable to execute due to failures in prior dependent apps. That is, an App that is dependent on the successful completion of another app will fail with a dependency error if any of the Apps on which it depends fail.

For Bash Apps, the `stdout` and `stderr` special keywords allow files to be configured for redirection of the script's standard output and standard error streams, respectively. An example Bash App is shown below.

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

As with Python Apps, Bash Apps are invoked using the standard Python syntax:

```
echo_hello().result()
with open('echo-hello.stdout', 'r') as f:
    print(f.read())
```

Parsl allows for common parallel programming patterns to be easily expressed using these simple building blocks. For example, massively parallel task execution can be achieved by repeatedly calling Apps in a loop, as is common in Monte Carlo simulations. Map-reduce execution can be added by collecting the futures from a loop (the map phase) and passing them as inputs to another App (the reduce phase); and complex dataflows can be constructed by chaining together a series of apps.

Listing 4 illustrates a dataflow utilizing two Python Apps. The `pi` App inserts a large number of random points into a square, determines whether each point is inside or outside a circle inscribed in the square, and uses this data to compute an estimate of pi. The workflow invokes several instances of the `pi` App which run asynchronously in parallel, then passes the corresponding futures to the `mean` App, which computes an average for the estimate of pi.

Listing 5 illustrates an example dataflow comprised of two Bash apps and a single Python App. The workflow starts multiple instances of the `generate` App asynchronously in parallel and produces output files, in this case containing random numbers. This step could represent a scientific simulation or analysis. The resulting files are then passed via futures to the Bash App `concat` that combines the generated files into a single file. Note that the Parsl script does not require explicit barriers or synchronization primitives, as Parsl delays execution of the `concat` App until all output files are created from the `generate` App. Finally, the Python `total` App parses the concatenated output file and computes the total by adding each line. At the conclusion of the workflow we insert a call to retrieve the `result` of the final future. This step ensures that the program waits for the workflow to complete.

Parsl objects include the Parsl file object. The Parsl file abstraction is important for enabling execution location independence of a Parsl program and avoiding hard-coding paths. Parsl files can be defined for local, HTTP, FTP, and Globus-accessible files. When a Parsl file is passed, Parsl's runtime uses a specialized data manager to transfer the file to where it is needed and to transparently translate the physical location of the file for Apps that need it.

Parsl supports modularization of sophisticated workflows in an intuitive, Pythonic way. Apps can be defined in libraries that are agnostic about the execution site and grouped by functionality. This has many benefits, including: 1) readability, 2) logical separation of components, and 3) reusability of components. The configuration(s) can be defined in a module or file saved independently, to be imported into the control script depending on which execution resources should be used. As an example, consider that the two configuration objects defined in Listing 1 and Listing 2 are saved to a file named `config.py`, and the following code is saved in a file called `library.py`.

```
from parsl.app import python_app

@python_app
def increment(x):
    return x + 1
```

```
import parsl
from parsl.app import python_app

parsl.load()

pts_per_estimate = 10**6
n_estimates = 10

@python_app
def pi(n_points):
    """Parsl app to estimate pi.

    Consider a circle of radius R inscribed inside a
    square of length 2r. The area of the circle is
    pi * r^2. The area of the square is (2r)^2. Thus, if
    `n_points` uniformly distributed random points
    are dropped within the square, then approximately
    `n_points` * pi / 4 will be inside the circle.
    """
    from random import random

    inside = 0
    for i in range(n_points):
        # Drop a random point in the box.
        x, y = random(), random()
        # Count points within the circle.
        if x**2 + y**2 < 1:
            inside += 1

    return (inside * 4 / n_points)


@python_app
def mean(inputs=[]):
    return sum(inputs) / len(inputs)

estimates = [pi(pts_per_estimate) for i in range(n_estimates)]

mean_pi  = mean(inputs=estimates)

print("Pi is approximately: {:.5f}".format(mean_pi.result()))
```

Listing 4: An example Parsl program which estimates the value of π using the Monte Carlo method. A number of separate estimates are run in parallel. Finally, the mean of the parallel computations is computed.

Further, consider the following control script, which imports the `increment` App. This file must import and load the configuration from `config.py` before calling the `increment` app from `library.py`:

```
import parsl
from config import thread_config
from library import increment

parsl.load(local_threads)

futures = [increment(i) for i in range(5)]
for i, f in zip(range(5), futures):
    print('{} + 1 = {}'.format(i, f.result()))
```

This will run in a local thread pool, and print

```python
import os

import parsl
from parsl.app import python_app, bash_app

parsl.load()

# App that generates a random number
@bash_app
def generate(outputs=[]):
    return "echo $(( RANDOM )) &> {outputs[0]}"

# App that concatenates input files into a single output file
@bash_app
def concat(inputs=[], outputs=[],
           stdout="stdout.txt", stderr='stderr.txt'):
    return "cat {} > {}".format(" ".join(inputs), outputs[0])

# App that calculates the sum of values
# in a list of input files
@python_app
def total(inputs=[]):
    total = 0
    with open(inputs[0], 'r') as f:
        for l in f:
            total += int(l)
    return total

# Create five files with random numbers
output_files = []
for i in range(5):
    path = os.path.join(os.getcwd(), 'random-{}.txt'.format(i))
    output_files.append(generate(outputs=[path]))

# Concatenate the files into a single file
inputs = [i.outputs[0] for i in output_files]
outputs = [os.path.join(os.getcwd(), 'all.txt')]
cc = concat(inputs=inputs, outputs=outputs)

# Calculate the sum of the random numbers
total = total(inputs=[cc.outputs[0]])
print(total.result())
```

Listing 5: An example Parsl program that generates random numbers in parallel, concatenates the results, and computes the sum of the random numbers.

```
0 + 1 = 1
1 + 1 = 2
2 + 1 = 3
3 + 1 = 4
4 + 1 = 5
```

To modify this program to run on the Stampede2 supercomputer, the developer would simply replace `thread_config` with `stampede_config`.

Developers often run the same workflow many times, with only incremental changes to each iteration. To avoid repeated execution of the same computations, developers may enable memoization of Parsl Apps. If enabled, a cache of the name, arguments, and function

```python
import os

import parsl
from parsl.app import bash_app
from parsl.data_provider.files import File

parsl.load()

# App that copies the contents of
# one or more files to another file
@bash_app
def copy(inputs=[], outputs=[]):
     return 'cat %s &> %s' % (inputs[0], outputs[0])

# Create a test file
path = os.path.join(os.getcwd(), 'in.txt')
with open(path, 'w') as f:
    f.write('Hello World!\n')

# Create Parsl file objects
infile = File(os.path.join(os.getcwd(), 'in.txt'))
outfile = File(os.path.join(os.getcwd(), 'out.txt'))

# Call the copy app with the Parsl file
future = copy(inputs=[infile], outputs=[outfile])

# Read what was redirected to the output file
with open(future.outputs[0].result(), 'r') as f:
    print(f.read())
```

Listing 6: An example Parsl program which uses the Parsl file abstraction.

```python
import parsl
from parsl.app import python_app

parsl.load()

@python_app
def sort_numbers(inputs=[]):
    with open(inputs[0].filepath, 'r') as f:
        strings = [n.strip() for n in f.readlines()]
        strings.sort()
        return strings

unsorted_file = File(
    'https://raw.githubusercontent.com/'
    'Parsl/parsl-tutorial/master/input/unsorted.txt'
)

f = sort_numbers(inputs=[unsorted_file])
print(f.result())
```

Listing 7: An example Parsl program which uses the Parsl file abstraction to access a remote file.

body will be maintained. App caching can be enabled by setting the cache argument in the python_app or bash_app decorator to True (by default it is False). App caching can be globally enabled by setting app_cache=True in the configuration. An example demonstrating App caching is shown in Listing 8.

Parsl implements checkpointing, which enables the workflow state to be saved and then used at a later time to resume execution from that point. Checkpointing facilitates recovery in the event of failure of the Parsl control process. Parsl checkpointing is incremental, i.e., each explicit checkpoint saves state changes from the previous checkpoint. Thus, the full history of a workflow can be distributed across multiple checkpoints.

4 EVALUATION

We have investigated Parsl scaling using the HTEX and EXEX executors on the XE component of Blue Waters at NCSA [4], a 13-petaflop Cray XE/XK hybrid system comprising 22,636 XE compute nodes (362,240 cores) and 4,288 XK compute nodes (33,792 cores) with an additional 4,228 Kepler Accelerators.

Figure 1 shows weak scaling results when executing 10 tasks per core for 10-sec and 100-sec tasks. For each executor and task duration, we deployed a worker per core on each node. We show results when increasing the number of nodes from 128 to 2,048 (~64K workers) for 10-sec tasks and from 128 to 8,192 (~256K workers) for 100-sec tasks. The results show that Parsl executors scale with the number of workers and tasks without significant performance degradation. Parsl scales to 2,048 nodes (~640K tasks) with HTEX and 8,192 nodes with EXEX (~2.56M tasks).

Figure 1 also illustrates the overhead of task submission: in order to scale efficiently we require longer duration tasks in order to keep cores busy. For good performance we estimate a need for *task duration / workers* ≥ 0.01. For example, when using 10K cores, it is best to use tasks that are at least 100 sec.

We also explored the scalability of Parsl when compared with alternative Python-based libraries: IPyParallel, Dask distributed, and FireWorks. We found that Dask distributed performed slightly better than HTEX and EXEX when there are fewer than 256 workers. IPyParallel and FireWorks had significantly worse scaling performance. The maximum number of nodes (and workers) we were able to scale to were: IPyParallel 64 nodes (2,048 workers); FireWorks 32 nodes (1,024 workers); and Dask distributed 128 nodes (4,096 workers). These results highlight the ability of Parsl to scale to very large systems.

5 USE CASES

Parsl has been used in a variety of scientific domains including biology, cosmology, materials science, chemistry, and social science. Two examples of its ongoing use are in a parallel scientific workflow for simulating cosmological images and as the basis for a high performance machine learning inference system.

Cosmology: In preparation for the imminent arrival of data from the Large Synoptic Survey Telescope (LSST), the Dark Energy Science Collaboration (DESC) is using Parsl to simulate raw exposures from the telescope. The workflow relies on the imSim software package [9] to construct images based on catalogs of astronomical objects, taking into account systematic effects of the atmosphere, optics, and telescope sensors. The workflow coordinates the execution of imSim for each sensor using input instance catalogs. Parsl manages the invocation of imSim, using Singularity containers to encapsulate the entire software stack. The Parsl workflow is responsible for processing instance catalogs, determining how to pack simulation workloads onto compute nodes, and invoking the Singularity containers deployed to each node. The workflow involves huge amounts of data: it simulates each of the LSST's 189 telescope sensors observing tens of thousands of instance catalogs, some containing millions of astronomical objects. The simulation

```
import parsl
from parsl.app import python_app, bash_app
from parsl.providers import LocalProvider
from parsl.channels import LocalChannel
from parsl.config import Config
from parsl.executors import HighThroughputExecutor

local_htex = Config(
    executors=[
        HighThroughputExecutor(
            cores_per_worker=1,
            provider=LocalProvider(
                init_blocks=1,
                max_blocks=10,
            )
        )
    ],
)

parsl.load(local_htex)

@python_app(cache=True)
def slow_message(message):
    import time
    time.sleep(5)
    return message

# First call to slow_message will calculate the value
first = slow_message("Hello World!")
print("First: {}".format(first.result()))

# Second call to slow_message with the same arguments
# will return immediately
second = slow_message("Hello World!")
print("Second: {}".format(second.result()))

# Third call to slow_message with different arguments
# will again wait
third = slow_message("Greetings, World!")
print("Third: {}".format(third.result()))
```

Listing 8: An example Parsl program that shows two calls to the slow message app with the same message. The first call to the app takes five seconds. Because App caching is enabled, the second call returns immediately, as the result is retrieved from the memoization table instead of being computed. Finally, the third call again takes five seconds, because it is called with a different argument.

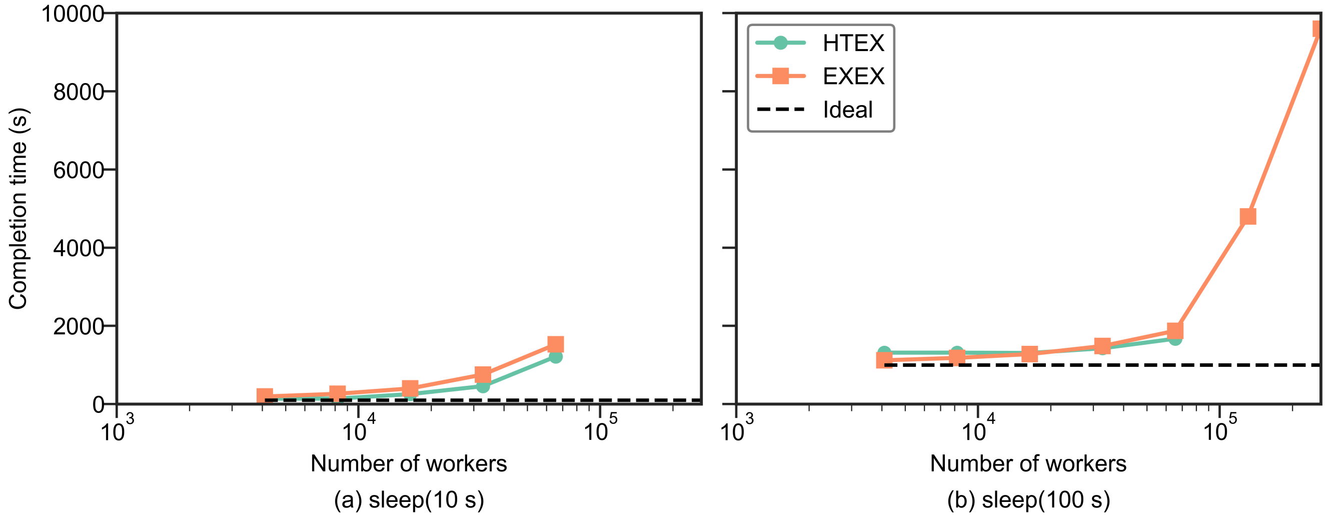

Figure 1: Weak scaling results when executing 10 tasks per core for 10-sec and 100-sec tasks

workflow has consumed more than 30M core hours to-date, using 4K nodes (256K cores) on Argonne's Theta supercomputer and 2K nodes (128K cores) on NERSC's Cori supercomputer, each for several days.

DLHub: The Data and Learning Hub for Science (DLHub) [7] provides the ability for researchers to publish, share, and invoke machine learning models. It allows researchers to publish their model by depositing and describing them in a catalog, obtaining a persistent identifier, and setting access permissions. Others can then discover, reuse, and cite the published models. In order to enable invocation of models, DLHub includes a Parsl-based ML inference engine that is optimized for executing large numbers of bag-of-tasks applications. DLHub provides a web service interface through which users can invoke a model on a set of input data (e.g., primitive types or files). DLHub uses Parsl to manage an elastic pool of resources (using Kubernetes) on which inference tasks are executed in containers. Parsl makes it easy to manage the parallel invocation of these tasks, scaling to thousands of connected workers and offering low-latency invocation.

6 RELATED WORK

There are a number of workflow systems that support the orchestrated execution of task dependency graphs. Many, such as Pegasus [12] and Galaxy [14] implement a static DAG model which is defined (often in XML) and then executed. Other approaches such as Swift [17], Swift/T [18], and NextFlow [13] implement their own languages for expressing workflows. Unlike Parsl, these approaches do not provide an intuitive and integrated way of parallelizing Python code.

Python-based workflow systems such as FireWorks [15], Apache Airflow [1], and Luigi [16] offer a mechanism for expressing workflow graphs in Python. However, each is designed for a different purpose than Parsl. For example, FireWorks focuses on providing reliable execution of longer running tasks, Airflow implements a model for executing task graphs expressed in Python, and Luigi introspects Python classes to derive connected components.

Dask [10] is Python library designed to support parallel data analytics. It offers parallel implementations of common Python libraries (e.g., NumPy and Pandas) that can replace their non-parallel equivalents. Dask distributed [11] implements a general model for integrating distributed execution on external clusters. It implements a scheduler that can manage execution across nodes in a cluster. Like Parsl, Dask's underlying APIs provide for wrapping of functions and asynchronous invocations using futures. However, Parsl offers various additional features such as flexible executor models, increased scalability to the largest supercomputers, and support for external applications and wide area transfers.

7 SUMMARY

Parsl addresses the growing need in science to provide an easy-to-use means of developing and executing parallel programs in Python, and specifically, those that are composed of various components. Parsl's innate flexibility and scalability allows for seamless portability and sharing of Parsl programs, allowing researchers to develop programs at small scale using threads, execute the same program on a campus cluster using a pilot job model, and then scale that program to thousands of nodes on a supercomputer, all by modifying only a single configuration object. Initial adoption of Parsl has been encouraging, with successful use cases in biology, materials science, social science, and computational chemistry, to name just a few. Further, evaluation of Parsl's performance highlights that it can significantly outperform comparable parallelism libraries in Python, scaling to 256,000 workers.

Parsl is an open source project available on GitHub: https://github.com/Parsl/parsl. Community contributions are welcome.

ACKNOWLEDGMENT

This work was supported in part by NSF award ACI-1550588 and DOE contract DE-AC02-06CH11357. This work made use of Blue

Waters, which is supported by the National Science Foundation (awards OCI-0725070 and ACI-1238993) and the state of Illinois.

REFERENCES

[1] Airflow Project. Airflow. https://airflow.apache.org/. Accessed Feb 1, 2019.

[2] Y. Babuji, K. Chard, I. Foster, D. S. Katz, M. Wilde, A. Woodard, and J. Wozniak. 2018. Parsl: Scalable Parallel Scripting in Python. In *10th International Workshop on Science Gateways (IWSG 2018)*. CEUR-WS.org. http://ceur-ws.org/Vol-2357/paper11.pdf

[3] Y. Babuji, A. Woodard, Z. Li, D. S. Katz, B. Clifford, R. Kumar, L. Lacinski, R. Chard, J. Wozniak, I. Foster, M. Wilde, and K. Chard. 2019. Parsl: Pervasive Parallel Programming in Python. In *28th ACM International Symposium on High-Performance Parallel and Distributed Computing (HPDC)*. ACM. https://doi.org/10.1145/3307681.3325400

[4] B. Bode, M. Butler, T. Dunning, T. Hoefler, W. Kramer, W. Gropp, and W.-m. Hwu. 2013. The Blue Waters super-system for super-science. In *Contemporary High Performance Computing*. Chapman and Hall/CRC, 339–366.

[5] K. M. Chandy and C. Kesselman. 1993. Compositional C++: Compositional parallel programming. In *Languages & Compilers for Parallel Computing*. Springer, 124–144.

[6] K. Chard, S. Tuecke, and I. Foster. 2014. Efficient and Secure Transfer, Synchronization, and Sharing of Big Data. *IEEE Cloud Computing* 1, 3 (Sep. 2014), 46–55. https://doi.org/10.1109/MCC.2014.52

[7] R. Chard, Z. Li, K. Chard, L. T. Ward, Y. N. Babuji, A. Woodard, S. Tuecke, B. Blaiszik, M. J. Franklin, and I. T. Foster. 2019. DLHub: Model and data serving for science. In *33rd IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE.

[8] Common Workflow Language. Common Workflow Language Specifications, v1.0.2. https://www.commonwl.org/v1.0/. Accessed Feb 1, 2019.

[9] Dark Energy Science Collaboration (DESC). imSim: GalSim based Large Synoptic Survey Telescope (LSST) image simulation package. https://github.com/LSSTDESC/imSim. Accessed Feb 1, 2019.

[10] Dask Development Team. Dask: Library for dynamic task scheduling. https://dask.org. Accessed Feb 1, 2019.

[11] Dask Development Team. Dask distributed. http://distributed.dask.org/en/latest/. Accessed Feb 1, 2019.

[12] E. Deelman, K. Vahi, G. Juve, M. Rynge, S. Callaghan, P. Maechling, R. Mayani, W. Chen, R. da Silva, M. Livny, et al. 2015. Pegasus, a workflow management system for science automation. *Future Generation Computer Systems* 46 (2015), 17–35.

[13] P. Di Tommaso, M. Chatzou, E. W. Floden, P. P. Barja, E. Palumbo, and C. Notredame. 2017. Nextflow enables reproducible computational workflows. *Nature Biotechnology* 35, 4 (2017), 316.

[14] J. Goecks, A. Nekrutenko, and J. Taylor. 2010. Galaxy: A comprehensive approach for supporting accessible, reproducible, and transparent computational research in the life sciences. *Genome Biology* 11, 8 (2010), R86.

[15] A. Jain, S. P. Ong, W. Chen, B. Medasani, X. Qu, M. Kocher, M. Brafman, G. Petretto, G.-M. Rignanese, G. Hautier, et al. 2015. FireWorks: A dynamic workflow system designed for high-throughput applications. *Concurrency and Computation: Practice and Experience* 27, 17 (2015), 5037–5059.

[16] Luigi Team. Luigi. https://github.com/spotify/luigi. Accessed Feb 1, 2019.

[17] M. Wilde, M. Hategan, J. M. Wozniak, B. Clifford, D. S. Katz, and I. Foster. 2011. Swift: A language for distributed parallel scripting. *Parallel Comput.* 37, 9 (2011), 633–652.

[18] J. M. Wozniak, T. G. Armstrong, M. Wilde, D. S. Katz, E. Lusk, and I. T. Foster. 2013. Swift/t: Large-scale application composition via distributed-memory dataflow processing. In *13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing (CCGrid)*. IEEE, 95–102.