Introducing Parsl: A Python Parallel Scripting Library

Yadu Babuji*, Alison Brizius*, Kyle Chard*, Ian Foster*, Daniel S. Katz§, Michael Wilde* and Justin Wozniak*
*Computation Institute, University of Chicago & Argonne National Laboratory, Chicago, IL, USA
§National Center for Supercomputing Applications, University of Illinois Urbana-Champaign, Urbana, IL, USA

***Abstract*—Researchers frequently rely on large-scale and domain-specific workflows to conduct their science. These workflows may integrate a variety of independent software functions and external applications. However, developing and executing such workflows can be difficult, requiring complex orchestration and management of applications and data as well as customization for specific execution environments. Parsl (Parallel Scripting Library), a Python library for programming and executing data-oriented workflows in parallel, addresses these problems. Developers simply annotate a Python script with Parsl directives; Parsl manages the execution of the script on clusters, clouds, grids, and other resources. Parsl orchestrates required data movement and manages the execution of Python functions and external applications in parallel. In this abstract we describe Parsl's architecture and highlight two domains in which it has been used.**

I. Introduction

Parsl is a Python-based parallel scripting library that supports asynchronous and implicitly parallel data-oriented workflows. Building on the model used by Swift [1], Parsl brings advanced parallel workflow capabilities to scripts (or applications) written in Python. Parsl scripts allow selected Python functions and external applications (called *Apps*) to be connected by shared input/output data objects into flexible parallel workflows. Parsl abstracts the specific execution environment, allowing the same script to be executed on multicore processors, clusters, clouds, and supercomputers.

When a Parsl script is executed, the Parsl library causes annotated functions (Apps) to be intercepted by the Parsl execution fabric, which captures and serializes their parameters, analyzes their dependencies, and runs them on selected resources (*sites*). The execution fabric brings dependency awareness to Apps by introducing data *futures.* If one App is responsible for writing a future, other Apps are blocked from reading it until it is written. This feature allows Apps to execute in parallel whenever they do not share dependencies or their data dependencies have been resolved.

This abstract describes Parsl, highlighting how it allows standard Python scripts to be augmented to represent complex workflows and facilitate parallel execution. We use two example workflows, from computational chemistry and biology, to highlight the power of the approach.

II. Parsl Model

The Parsl architecture is shown in Fig. 1. Parsl scripts are decomposed into a simple dependency graph by the *DataFlow Kernel* (DFK). The DFK manages execution of individual Parsl Apps on a variety of sites. Unlike parallel scripting languages like Swift, in which every variable and piece of code is asynchronous, Parsl relies on user annotations and futures to specify and manage concurrency. At present, Parsl provides a lightweight data management layer in which files are staged to the execution site via a dedicated communication channel.

Execution: The DFK provides a lightweight abstraction over different execution resources. This abstraction is at the heart of Parsl's ability to transparently support different execution fabrics.

Parsl launches asynchronous Apps and passes futures to other Apps in lieu of computing results synchronously. The DFK is responsible for managing a script's execution, making ordinary functions aware of futures and ensuring the execution of these functions are conditional on the resolution of all dependent futures. This enables completely asynchronous management of all launched tasks with the data dependencies alone determining the order of execution.

When instantiating the DFK, developers specify the specific *execution providers* and *executors* that will be used for executing the parallel components of the script. Execution providers are simple abstractions over computational resources. Executors provide an abstraction layer for executing tasks. At present, Parsl supports three different executors: threads, Swift/T [2], and IPythonParallel.

Apps: A Parsl script is comprised of standard Python code plus a number of Apps—annotated units of Python code or external applications that specify their input and output characteristics and that may be run in parallel. An App may be defined by wrapping an existing function or the execution of an external command-line application using Bash scripting with the `@App` decorator. Listing 2 shows examples of these two types of Parsl Apps.

Futures: Parsl Apps are completely asynchronous. When an App is invoked there is no guarantee of when the result will be returned. Instead of directly returning a result, Parsl returns an *AppFuture*: a construct that includes the real result as well as the status and exceptions for that asynchronous function invocation. Parsl also supplies methods to examine the future construct, including a status check, blocking on completion, and retrieving results. Parsl leverages Python's `concurrent.futures` module for this purpose.

Parsl also introduces a model for managing file-based futures. Such *DataFutures* represent the asynchronous output files generated by an App invocation. DataFutures extend the AppFuture model by providing support for a range of operations related to files.

III. Case Studies

We present two workflows written in Parsl to illustrate how it can satisfy the needs of different application domains.

SwiftSeq [3] is a bioinformatics workflow that supports aligning and genotyping gene panels, exomes, and whole

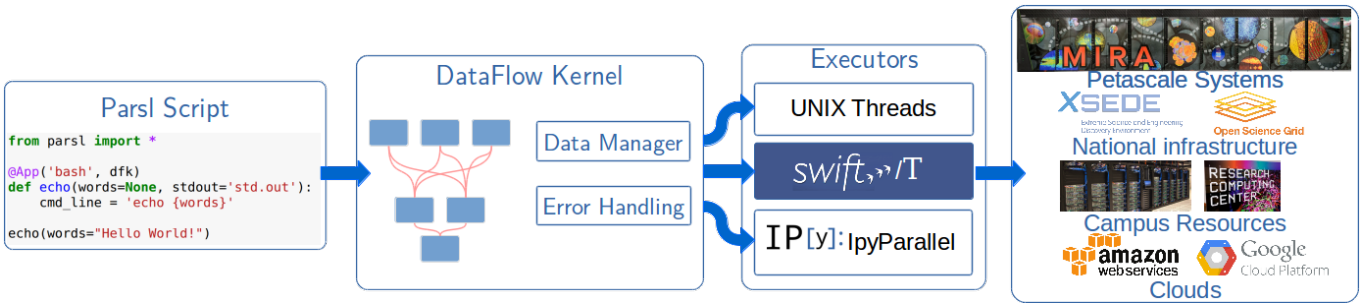

Fig. 1: Parsl architecture. The DataFlow Kernel maps scripts to Executors that support diverse computational platforms.

```
@App('python', dataflowkernel)
def hello():
    return 'Hello World!'

@App('bash', dataflowkernel)
def hello(outputs=[], stdout=None,
          stderr=None):
    cmd_line = '''echo "Hello World" '''
```

Fig. 2: Two examples of Parsl Apps

genomes. The workflow is comprised of approximately 10 applications that communicate by writing and reading files. While applications must often execute in sequence, there are also opportunities for parallelism. First, the workflow is often executed on many samples, each of which can be analyzed in parallel; second, the large genetic sequences can be divided up and analyzed in parallel; and finally, some of the applications can also be executed in parallel. SwiftSeq benefits not only from Parsl's ability to specify such parallelism, but also from its ability to express a complex workflow, manage the flow of data between applications, recover from errors, and execute on a variety of computational resources.

The second example workflow is a molecular dynamics workflow that uses PACKMOL [4] to assemble initial starting configurations of ionic liquid molecules with a protein (e.g., Trp-cage), and then subsequently energy minimizes, heats, equilibrates, and runs production molecular dynamics simulations using the GPU-accelerated version of Amber [5]. The workflow relies on three separate applications that are executed iteratively to perform different functions. PACKMOL is used to generate the system configuration; AmberTools are used to create input coordinate and parameter files for simulations; and Amber is used to run various simulations. Parsl allows a wide range of different system configurations to be considered in parallel, and it also allows simple error handling logic to be expressed.

IV. Related Work

Many workflow systems have been developed to facilitate the expression and execution of arbitrary, data-oriented workflows, for example, the Swift parallel scripting language. A weakness of these systems, however, is their lack of consistency with typical research development environments.

There is increasing interest in developing Python-based workflow tools that better match common research environments, e.g., Dask [6], Apache Airflow [7], and Luigi [8]. These systems enable Python developers to author and execute workflows, using very different models. Dask focuses on parallel analytics, providing Dask-specific modules to be used in place of common analytics modules (e.g., Dask Dataframe in place of Pandas Dataframe) to facilitate parallel execution. Airflow allows developers to create an explicit graph of independent tasks, including specifying the relationships between those tasks. Luigi allows developers to create a pipeline of batch jobs for submission to a scheduler. Parsl is differentiated by its focus on enabling annotation of existing Python scripts and providing implicit parallelization of annotated Apps.

V. Summary

Parsl provides an easy-to-use model for developing Python-based workflows that include arbitrary Python functions and external applications. It abstracts the complexity of interacting with different resource fabrics and instead supports the development of resource-independent Python scripts. While development of Parsl has been ongoing for less than a year, we build upon a decade of experience developing the Swift parallel scripting language and aim to bring these same capabilities to Python. Already, Parsl has been used to create successful workflows in biology and chemistry, and it is being tested in materials science and astronomy.

Acknowledgment

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

References

[1] M. Wilde, M. Hategan *et al.*, "Swift: A language for distributed parallel scripting," *Parallel Computing*, vol. 37, no. 9, pp. 633–652, Sep. 2011.
[2] T. G. Armstrong, J. M. Wozniak *et al.*, "Compiler techniques for massively scalable implicit task parallelism," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '14, 2014, pp. 299–310.
[3] J. Pitt, "Swiftseq," http://www.igsb.org/software/swiftseq.
[4] L. Martínez, R. Andrade *et al.*, "PACKMOL: A package for building initial configurations for molecular dynamics simulations," *J. Comp. Chemistry*, vol. 30, no. 13, pp. 2157–2164, 2009.
[5] D. A. Case, T. E. Cheatham *et al.*, "The Amber biomolecular simulation programs," *J. Comp. Chemistry*, vol. 26, no. 16, pp. 1668–1688, 2005.
[6] M. Rocklin, "Dask: Parallel computation with blocked algorithms and task scheduling," in *Proc. 14th Python in Sci. Conf.*, 2015, pp. 130–136.
[7] Apache Airflow Project, "Apache Airflow," https://airflow.incubator.apache.org/.
[8] Spotify, "Luigi," https://github.com/spotify/luigi.