Parsl: Pervasive Parallel Programming in Python

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

Rohan Kumar
University of Chicago
rohankumar@uchicago.edu

Lukasz Lacinski
University of Chicago
lukasz@uchicago.edu

Ryan Chard
Argonne National Laboratory
rchard@anl.gov

Justin M. Wozniak
Argonne National Laboratory
woz@anl.gov

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

Michael Wilde
ParallelWorks
wilde@parallelworks.com

Kyle Chard
University of Chicago
chard@uchicago.edu

ABSTRACT

High-level programming languages such as Python are increasingly used to provide intuitive interfaces to libraries written in lower-level languages and for assembling applications from various components. This migration towards orchestration rather than implementation, coupled with the growing need for parallel computing (e.g., due to big data and the end of Moore's law), necessitates rethinking how parallelism is expressed in programs. Here, we present Parsl, a parallel scripting library that augments Python with simple, scalable, and flexible constructs for encoding parallelism. These constructs allow Parsl to construct a dynamic dependency graph of components that it can then execute efficiently on one or many processors. Parsl is designed for scalability, with an extensible set of executors tailored to different use cases, such as low-latency, high-throughput, or extreme-scale execution. We show, via experiments on the Blue Waters supercomputer, that Parsl executors can allow Python scripts to execute components with as little as 5 ms of overhead, scale to more than 250 000 workers across more than 8000 nodes, and process upward of 1200 tasks per second. Other Parsl features simplify the construction and execution of composite programs by supporting elastic provisioning and scaling of infrastructure, fault-tolerant execution, and integrated wide-area data management. We show that these capabilities satisfy the needs of many-task, interactive, online, and machine learning applications in fields such as biology, cosmology, and materials science.

CCS CONCEPTS

• **Computing methodologies** → **Parallel computing methodologies**; **Distributed computing methodologies**; • **Software and its engineering** → *Software libraries and repositories.*

KEYWORDS

Parsl; parallel programming; Python

1 INTRODUCTION

The past decade has seen a major transformation in the nature of programming. Software is increasingly constructed by using a high-level language to integrate components from many sources. In other words, much software is not so much written as assembled. Additionally, as data sizes increase and sequential processing power plateaus there is a growing need to make use of parallel hardware such as specialized accelerators and distributed computing systems. As a contribution to a merging of these two trends, we present here methods that allow for the natural expression of parallelism within a popular high-level language, Python, in such a way that programs can express opportunities for parallelism that can then be realized, at execution time, using different execution models on different parallel platforms.

Specifically, we present Parsl [16], a parallel scripting library that defines and implements Python decorators that developers can use to express parallelism with Python programs. We show how programmers can thus easily create programs composed of both Python functions and components written in other languages, with opportunities for parallel execution expressed in a way that allows for efficient implementation on a variety of architectures. Parsl

thus implements a form of compositionality, in which a program is constructed by composing component programs in parallel. This approach to parallelism is in contrast to prior efforts that rely on domain-specific languages (DSL) [25, 27, 39], configuration-based models [1, 24, 31], Python-based graph descriptions [4, 33], and compiled language extensions [19] to support such composition. We choose to extend Python for several reasons: it is widely used for programming; as an interpreted language it can be easily extended (without requiring compiler modifications); and while not considered a functional programming language, it lends itself well to a functional style that fits our model of parallelism.

Parsl enables a simple functional programming model at the task level while still retaining procedural Python code for other aspects of the program. That is, it allows developers to declare the logic of a program, without explicitly describing how components are executed, by chaining together functions with defined input and output objects. Developers simply annotate functions in a Python script as *Apps*, indicating that these functions (either pure Python or components in other languages) can be executed concurrently, if permitted by data dependencies. Parsl composes a dynamic dependency graph and maps the task graph to arbitrary execution resources, exploiting parallelism where possible by tracking the creation of data (objects or files) and determining when App dependencies are met. The explicit input/output specification for Apps, immutable input and output object passing, and dependency-based execution model together enable users to construct safe and deterministic parallel programs. Parsl is inherently flexible and scalable. Its modular execution architecture supports a variety of execution models—from pilot jobs to extreme scale distributed execution—and a variety of execution providers, from laptops to supercomputers. By separating code from configuration, Parsl allows parallelism to be realized in different ways on different resources, without changing program code. Together, these features enable simple, safe, scalable, and flexible parallelism.

The fact that Parsl extends Python offers many advantages over alternative approaches. For example, Parsl can be easily configured on a variety of computers (including in virtual Python environments), as Python is part of standard distributions and is already deployed in many environments. Parsl allows users to express parallelism in familiar Python code, without needing to learn new syntax. Developers familiar with Python can, in a matter of minutes, learn the additional constructs offered by Parsl. Furthermore, Parsl programmers are not constrained to using only Parsl constructs: they can access the complete power of Python to implement sequential functionality (e.g., argument parsing, logging, visualization libraries), leverage the vibrant and diverse scientific Python (SciPy) ecosystem (e.g., Pandas dataframes, Scikit-learn, Jupyter notebooks, and Python interfaces to other tools like Tensorflow), and exploit powerful language features such as conditional and loop constructs, complex datatypes, generators, list comprehensions, and other object-oriented functionality. This flexibility is important because there is usually much more to a program than parallelism. Parsl embraces this idea, making it easy to augment Python programs with Parsl-based parallel components, rather than relying on complex nesting or specifications defined in terms of markup languages.

In this paper we describe our motivation for developing Parsl, the design decisions that influenced its design, and its implementation architecture. We evaluate the scalability of Parsl on a campus cluster and a supercomputer, showing that its high-throughput executor can scale to more than 65 000 concurrent workers with throughput greater than 1000 tasks/second, that its extreme-scale executor can scale to more than 250 000 concurrent workers over 8000 nodes, and that its low-latency executor can execute components within 5ms, far exceeding what other Python-based systems can achieve.

This paper is structured as follows. In §2 we outline five use cases that motivated the development of Parsl. In §3 and §4 we outline the design and architecture of Parsl, respectively. In §5 we evaluate the scalability and performance of Parsl. We present related work in §6. Finally, in §7 we summarize our contributions.

2 MOTIVATION

To motivate our approach we first highlight five scientific use cases that exhibit a variety of requirements, including the management of progress, concurrency, and data. We then describe the advantages of implementing Parsl as an extension to Python.

2.1 Use Cases

Parsl has been designed to support a broad range of science and engineering use cases, from traditional many-task computing to new computational modes such as interactive computing (e.g., in Jupyter notebooks), machine learning (training and inference), and online computing. Here we outline several of the scientific use cases that collectively influenced Parsl's architecture, and we summarize the requirements for each use case in Table 1.

A common many-task application is DNA **sequence analysis**, which is computationally-intensive, data-intensive, and requires multiple processing steps using various processing tools (e.g., alignment, quality control, variant calling, etc.) One specific DNA sequencing analysis application is SwiftSeq [37], which combines many processing tools and performs highly parallel execution on clouds, clusters, and supercomputers. SwiftSeq allows researchers to specify analysis requirements (e.g., analysis type, tools to be used, tool parameters) and then dynamically generates a many-task workflow for processing. SwiftSeq is implemented in Python, and requires a simple way of expressing parallelism for its processing tools and sequencing data. It is intended for analysis of thousands of genomes, each several GB in size, and with processing tools that run for minutes to hours. These long-running tools require fault tolerance. Efficient use of infrastructure requires that individual tasks be partitioned, where possible, and placed across multiple nodes and even sites.

An example of a bag-of-tasks application is **ML inference**. Increasingly, science is performed through multi-tenant, service-oriented platforms, such as scientific portals and gateways. These services provide on-demand access to the data, tools, and infrastructure required for thousands of researchers to concurrently perform analyses. The Data and Learning Hub for science (DLHub) [21] is one such service designed to publish and serve ML models for parallel inference by a community of researchers. DLHub requires methods to manage many short-duration inference requests using a bag-of-tasks execution model. Unlike other bag-of-tasks workloads,

Table 1: Requirements from five different scientific use cases. HTC=High Throughput Computing; FaaS=Function as a Service.

	Sequence analysis	ML inference	Materials science	Neuroscience	Cosmology
Pattern	dataflow	bag-of-tasks	dataflow	sequential	dataflow
Paradigm	HTC	FaaS	Interactive	Batch	HTC
O(Nodes)	hundreds	tens	tens	tens	thousands
O(Tasks)	thousands	thousands	hundreds	hundreds	millions
O(Duration)	days	seconds	minutes	hours	day
O(Data)	TB	KB	MB	GB	TB
Latency Sensitive	no	yes	yes	no	no

DLHub is used in a variety of real-time workloads that require low-latency responses for example to detect errors. Finally, task isolation, via containers, is desirable to accommodate the vanguard requirements of diverse ML models.

An example of interactive computing in **materials science** is modeling stopping power in Jupyter notebooks. Because traditional time-dependent density functional theory (TD-DFT) computations are expensive, researchers are developing machine learning methods to augment TD-DFT simulations. For example, by using existing DFT data stored in the Materials Data Facility (MDF) [17] to create surrogate models that predict stopping power in different directions. During the model development phase, researchers use Jupyter notebooks to iteratively develop and evaluate their models. They therefore require methods that make it easy to parallelize these processes, low-latency responses when exploring modeling approaches in an interactive manner, and scalability to exploit HPC resources for final model execution.

Experimental research processes often integrate computational analyses for error checking, quality control, and experiment steering. For example, **neuroscience** researchers use x-ray microtomography at the Advanced Photon Source to characterize the neuroanatomical structure of brain volumes to study brain aging and disease [22]. They combine analysis processes, in near-real time, to reconstruct 3-dimensional images for error detection and sample orientation during the experiment. Such analyses are implemented by first identifying the center of imaged samples from amongst hundreds of 2-dimensional slices, applying a machine learning model to identify the highest quality slices, and then using tomographic reconstruction to create a 3D model from the slices. To reliably perform such reconstructions in a timely manner, researchers must be able to leverage the largest computing resources available to them, such as those at Argonne's Leadership Computing Facility.

Analyses can also have unpredictable performance. An example is a **cosmology** simulation that aims to produce simulated images from the Large Synoptic Survey Telescope (LSST). To create these simulated images, researchers first construct more than 10 000 instance catalogs of cosmological objects using observation parameters (e.g., time, altitude, temperature, telescope configuration) and astrophysical inputs (e.g., stars, galaxies). They then use these catalogs to simulate images acquired from each of the LSST's 189 sensors. To simplify development and portability, the simulation code is written in Python and packaged in Singularity containers. Given that the simulation can occupy the full capacity of a supercomputer for weeks, the parallel execution system must be capable of maintaining high utilization at scale. For example, as execution time is dependent on the number of objects included in a sensor/-catalog, there is potential for significant imbalance throughout the simulation, thus the simulation must group (and rebalance) tasks into appropriate sized bundles for a given processing node (e.g., 64 tasks for a 64-core processor).

2.2 Why build on Python?

Python, first released in 1991, has become the **lingua franca** in many domains. Many developers, scientists, and analysts use Python extensively as it is straightforward to learn, well documented, and reliable. Python is a powerful programming language with sophisticated features that need neither be reimplemented by Parsl nor relearned by programmers: for example, if statements, loops, generators, objects, and list comprehensions. Python also has a rich and vibrant ecosystem with many useful libraries.

We briefly outline here some of the advantages of implementing Parsl as a library for Python.

Parsl semantics differ from those of Python only where necessary. Everything else stays the same and need not be designed, implemented, or learned. Thus, Parsl is familiar to Python developers and can be easily learned by others. Although Parsl focuses on task-oriented parallel computation, a program needs to do other things—often relatively trivial booking-keeping work that should rightly be implemented easily. As a tradeoff, reasoning about the program as a whole can be harder (affecting the correctness of checkpointing and retries, for example).

A single language for the implementation of Parsl and for writing programs. Parsl blurs the line between library and programs written with the library. For example, the LSST simulation above can use a small, straightforward piece of Python code to rate limit and rewrite the program's work queue and thus influence Parsl's scheduling in a non-trivial, program-specific manner. Such behavior is not expressible as part of the task dependency graph; but being application specific, neither would it be appropriate for it to be part of the core Parsl library.

Programming language features are beneficial for programming in the large. Like any other Python code, Parsl can be used to create programs, and standard libraries of Parsl Apps for particular domains can be developed and shared. These libraries can then be easily be composed with other code to create new programs. The Parsl runtime can execute code from many such libraries, keeping all the benefits of Parsl task execution for the program as a whole. Parsl apps look like Python function calls; asynchronously computed result values look like Python values and can be passed around and stored as such. Artificial barriers caused by one component of a program ending and another beginning can be avoided: as the whole program can be written in one language, the runtime can manage the execution of all tasks, no matter the component

from which those tasks originates, and can run tasks from different components in parallel on the same resource.

Programming language features are useful for programming in the small. As features develop, domain-specific languages designed to manage progress and concurrency may struggle to solve simple problems that are easily managed in sequential scripting languages. For example, a DAG-based language may need to choose a later action based on an earlier result. In a generic imperative programming language, a simple `if` statement suffices; a DAG-based approach to the same problem may require generating workflows inside other workflows, or embedding a richer language deep inside the outer DAG description. Similarly, a loop over a dynamically generated dataset can be accomplished with a simple `for` statement in Python; in a DAG-based language, it might require a separate workflow generation stage that cannot be scheduled as part of the main workflow. It is access to these Python language constructs that contributes to Parsl's generality and learnability.

3 DESIGN

Parsl is designed to enable intuitive parallel and compositional programming in Python that address broad scientific use cases, such as those outlined in the previous section. Specifically, we focus on five core design challenges: enabling the intuitive description of parallelism in Python; decomposing dataflow dependencies into a dynamic task graph for efficient execution; abstracting task execution mechanisms and environments to enable portability; supporting execution of heterogeneous workloads that range from short-running to long-running tasks, from few to millions of tasks, and from small-scale to large-scale resources; and enabling reliable parallel program execution.

3.1 Programming with Parsl

Parsl is designed for Python. When facing design decisions, we have chosen to remain as close to standard Python as possible, ensuring that Parsl is easy to learn and that Parsl scripts remain Pythonic.

At the core of Parsl are two constructs that introduce asynchronicity into Python: The App decorator and `future` object that can be used together to compose programs. These constructs allow Python functions to be executed asynchronously, in parallel, and potentially in a different execution location.

3.1.1 Apps. Parsl uses decorators to intercept and modify the behavior of Python functions. In Parsl two kinds of decorators are supported: the `@python_app` decorator for pure Python functions and `@bash_app` decorator for shell commands.

When either a Python App or Bash App is invoked, an asynchronous task is registered with Parsl and a `future` object is returned immediately, in lieu of the results of the computation. Eventually Parsl executes the task and results are made available through the `future`. To guarantee safety in a concurrent setting, Parsl Apps must be pure functions, acting only on their input arguments. The following is a minimal example of a Python App.

```
@python_app
def hello1(name):
    return "Hello {}".format(name)
```

As for Bash Apps, the Python code that forms the body of the function should return a fragment of Bash shell code. That shell code will be executed in a sandbox environment. Input and output handling behaves as with Python apps, although the return value from Bash Apps are UNIX return codes that indicate only whether the code succeeded. The Bash App also can be further configured using special keywords that allow for redirection of STDOUT/STDERR streams to files. The following is an example.

```
@bash_app
def hello2(name, stdout=None, stderr=None):
    return "echo 'Hello {}'".format(name)
```

Apps of either type can be invoked with standard Python syntax:

```
f1 = hello1("World")
f2 = hello2("World")
```

Thus, Parsl App functions invoked this way automatically follow Parsl semantics, without Parsl syntax at the call site. This is achieved through Parsl's use of futures and a parallel runtime.

3.1.2 Futures. We saw earlier that invoking Parsl Apps return `futures`. A `future` is an object that can be used to access the results of an asynchronous computation. Parsl futures implement a synchronous blocking method `future.result()` that will wait until the computation has yielded results or raised an exception. A non-blocking method, `future.done()`, immediately returns a boolean that indicates the status of the execution.

Since a `future` is created and updated only by a specific App invocation, it acts as a single-update variable, as commonly used in task-parallel systems [19, 27, 28, 39]. Futures are the only synchronization primitive offered by Parsl.

3.2 Input and output data

Any input data required by the App must be passed as input parameters to the invocation. While this represents a familiar way for Python users to pass objects, we note that other ways of passing information to Python function invocations, such as global variables, cannot be used: a price paid to allow for easy movement of tasks among execution resources. Similarly, any output from the App invocation needs to be explicitly passed back—as a return value or as an output file.

The following three Python types can be passed as inputs or outputs, with caveats:

Python objects. Any Python object that can be serialized (e.g., using Python's pickle [5] or dill [34] libraries), can be passed as an input parameter. Most Python objects that represent "data" (rather than, for example, file descriptors or threads) can be serialized. Objects passed as inputs should be treated as immutable: their contents should not be modified on either the submitting or executing side, as non-deterministic behavior can result otherwise. Parsl does not attempt to provide any richer distributed object model.

Files are declared using a Parsl `File` object, which can represent a remote file using various protocols. Parsl stages in file inputs and translates paths transparently so that they are available in the runtime environment of the executing program. Parsl provides convenience keyword arguments `inputs` and `outputs` in App functions that allow developers to dynamically specify collections of input and output files. Code should not modify input files, and should

not make assumptions about visibility of modifications between App invocations: see §4.5.

Futures will eventually contain some value, which may be of any of the above types.

Parsl is not unnecessarily strict about enforcing these rules: for example, if a shared filesystem is available Parsl file staging is not necessary. However, if a programmer chooses to make that tradeoff, the resulting code may not be able to be executed in an arbitrary environment.

3.3 Compositionality

Parsl allows applications to be composed by passing futures between Apps. An arbitrary number of futures can be passed to an App as arguments, implicitly indicating dependencies on the asynchronous execution of all Apps whose futures were passed as arguments. Since Apps can be invoked asynchronously, an arbitrarily large task graph can be constructed with minimal compute cost. Parsl can then execute these tasks based on the available parallelism within the task graph.

3.4 Runtime

Parsl's runtime is responsible for managing the parallel execution of Parsl-annotated components in a program on configured resources. To do so, Parsl assembles and dynamically updates a task dependency graph. This graph contains all state for the program and can be efficiently introspected and mapped to available execution resources. The task graph is represented as a directed acyclic graph (DAG) in which the nodes represent tasks and the edges represent the input/output data exchanged between tasks. The advantage of using a dynamic task graph is twofold: first, the execution of tasks may start as soon as the first task is identified, rather than waiting for the entire task graph to be formed before execution; and second, it allows for complex logic to be implemented in the program (e.g., loops and conditionals) as well as for tasks to generate new tasks during execution.

3.5 Separation of Code and Configuration

Parsl separates program logic from execution configuration, with the latter described by a Python object so that developers can easily introspect permissible options, validate settings, and retrieve/edit configurations dynamically during execution. A configuration specifies details of the provider, executors, connection channel, allocation size, queues, durations, and data management options. Listing 1 illustrates a basic configuration for the Stampede2 supercomputer. This configuration uses the HighThroughputExecutor to submit tasks from a login node (LocalChannel). It requests an allocation of 128 nodes, from the skx-normal partition, for up to 12 hours.

Many use cases require the ability to mix-and-match execution resources in various ways. For example, using GPU nodes for GPU-optimized codes and CPU nodes for others; combining thread-based execution of lightweight tasks with cluster-based execution of larger tasks; or even executing tasks across a number of computing resources wherever allocations may be available. Parsl supports these use cases by enabling "multi-site" execution via specification of more than one executor in the configuration.

Listing 1: Parsl configuration for Stampede2.

```
config = Config(
    executors=[
        HighThroughputExecutor(
            label="stampede2_htex",
            address=address_by_hostname(),
            provider=SlurmProvider(
                channel=LocalChannel(),
                nodes_per_block=128,
                init_blocks=1,
                partition="skx-normal",
                walltime="12:00:00"
            )
        )
    ]
)
```

3.6 Scalable Execution

Parsl is designed to support a wide range of use cases, from few long tasks to millions of short tasks. A system designed to address any such use case individually is likely to be unsuitable for other use cases. We focus instead on providing a generic and extensible model for supporting these varying use cases.

Parsl's modular executor interface is designed to support different mechanisms for managing the execution of tasks. The executor controls the process by which the task is transported to configured resources, executed on that resource, and results are communicated back to Parsl. In §4.3 we describe a set of executors that provide high-throughput, low-latency, and extreme-scale execution.

As execution progresses the resources required can vary. For example, the map-reduce pattern, common in many-task workflows, starts with a large set of map tasks, the results of which are combined by a smaller number of reduce tasks. To minimize resource wastage, Parsl is designed to automatically provision and deprovision execution resources. This elasticity component is controlled by an extensible *strategy* module within Parsl. The strategy module tracks outstanding tasks and available capacity on connected executors. It communicates with the connected providers to automatically scale to match real-time requirements.

3.7 Fault-Tolerant Execution

A Parsl program may fail due to failure of one of its Apps or of a node used for execution. As analysis sizes increase, so too does the likelihood of failure. In order for Parsl to be usable, it must expect failures and respond accordingly. For example, when re-executing a branch of failed execution, a user is unlikely to want to re-execute another branch that completed successfully.

Parsl provides fault-tolerance at the level of programs rather than Apps. That is, it treats tasks as the basic unit for fault tolerance, enabling checkpointing of execution state whenever a task completes. This ensures that, if desired, a user may re-execute a program and any Apps that are called with the same arguments need not be re-executed. While Parsl provides no App-level fault-tolerance, it works with Apps that implement their own fault-tolerance, e.g., via

checkpointing; such methods are opaque to Parsl. A more integrated model of fault-tolerance is future work.

4 ARCHITECTURE AND IMPLEMENTATION

Figure 1 shows Parsl's high-level architecture. In this section we briefly outline its core components and capabilities.
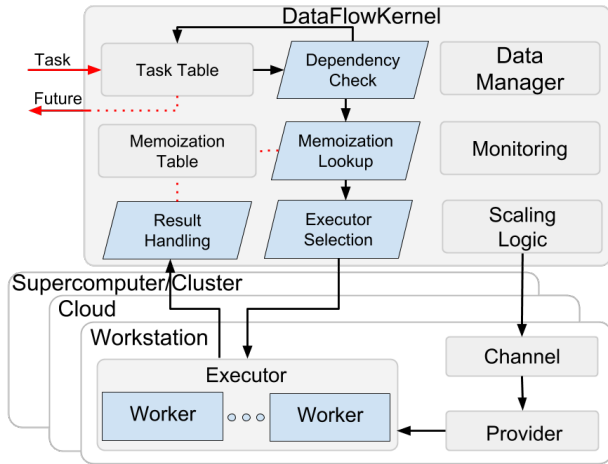

Figure 1: Parsl architecture. The DataFlowKernel stores the task graph and manages execution via one or more connected executors, of which three are shown here.

4.1 DataFlowKernel

The DataFlowKernel (DFK), Parsl's execution management engine, is responsible for constructing and orchestrating the execution of the task graph. DFK tracks task information (nodes in the task graph) in a Python data structure. Dependencies between Apps are implicitly derived from the passing of futures between Apps. Edges in the task graph are encoded as asynchronous callbacks on a dependent future, allowing DFK to be event driven, with each node in the task graph being considered upon resolution of its edges. Launching a task incurs a small fixed cost, as does triggering each outgoing edge when a task succeeds. Thus the execution time complexity of a task graph with n tasks and e edges is $O(n + e)$.

Once all of a task's dependencies have resolved successfully, DFK schedules the task for execution on a configured executor. If multiple executors are available, and the task contains no execution hints, an executor is picked at random. The executor responds with its own future that DFK associates with the future that was created when the App was invoked.

DFK tracks task state and waits for the future to be resolved, or a timeout period to elapse. In the case an App fails, as indicated by an exception in the resulting future or timeout, Parsl is able to retry the task by resubmitting it to an executor. If retries are disabled, or if the number of retries is exceeded, DFK wraps the exception (e.g., App execution errors or remote system failure) and associates it with the future. When memoization or checkpointing is used, DFK computes a hash of the App's function body and performs a lookup in a checkpoint file or memoization table using the function name, body hash, and arguments as the key. If the lookup succeeds, the result from the checkpoint file or memoization table is returned.

4.2 Providers

Clouds, supercomputers, and local PCs offer vastly different modes of access. To overcome these differences, and present a single uniform interface, Parsl implements a simple provider abstraction. This abstraction is key to Parsl's ability to enable scripts to be moved between resources. While there have been many prior efforts to implement standard interfaces for accessing various resources [29, 30], we chose to develop a lightweight abstraction entirely in Python for Parsl. The provider interface is based on three core actions: submit a job for execution (e.g., sbatch for the Slurm resource manager), retrieve the status of an allocation (e.g., squeue), and cancel a running job (e.g., scancel). Parsl implements providers for local execution (fork), for various cloud platforms using cloud-specific APIs, and for clusters and supercomputers that use a Local Resource Manager (LRM) to manage access to resources. Given the simplicity of the provider interface, it is easy to add new providers. Currently, Parsl implements providers for Slurm, Torque/PBS, HTCondor, Cobalt, GridEngine, AWS, Google Cloud, Jetstream, and Kubernetes.

Each provider implementation may allow users to specify additional parameters for further configuration. Parameters are generally mapped to LRM submission script or cloud API options. Examples of LRM-specific options are partition, wall clock time, scheduler options (e.g., #SBATCH arguments for Slurm), and worker initialization commands (e.g., loading a conda environment). Cloud parameters include access keys, instance type, and spot bid price.

4.2.1 Channels. Parsl scripts may be executed locally (e.g., on a login node or notebook server at an HPC center) or remotely (e.g., on a laptop, another cluster, or even a laptop). To support these different scenarios, Parsl introduces the notion of a *Channel* that describes how Parsl should authenticate and connect to the provider. Parsl includes two primary channels: LocalChannel for execution on a local resource, where the execution node has direct queue access, and SSHChannel, when executing remotely.

4.2.2 Launchers. Many LRMs offer mechanisms for spawning applications across nodes inside a single job and for specifying the resources and task placement information needed to execute that application at launch time. Common mechanisms include srun (for Slurm), aprun (for Crays), and mpirun (for MPI). The Parsl *Launcher* abstracts these system-specific launcher systems used to start workers across cores and nodes. Users may optionally specify a launcher in the provider configuration to control how Parsl communicates with the LRM. Parsl currently supports the following launchers: fork, srun, aprun, mpiexec, and GNU parallel.

4.2.3 Resource management. One significant challenge when designing a system that makes use of heterogeneous execution resource types is the need to provide a uniform representation of resources. Consider that single requests on clouds return individual nodes, clusters and supercomputers provide batches of nodes, grids provide cores, and workstations provide a single multicore node. To further complicate matters, some batch systems enforce policies that restrict the number of nodes permitted in a job (e.g., min, max, or in common groupings). From a scheduling perspective, the resources required by each task that is to be scheduled may range from a fraction of a node through to multiple nodes in the case of MPI applications, creating a bin-packing problem. Parsl defines

a resource unit abstraction called a *block* as the most basic unit of resources to be acquired from a provider. A block contains one or more nodes and maps to the different provider abstractions. In a cluster, a block corresponds to a single allocation request to a scheduler. In a cloud, a block corresponds to a single API request for one or more instances. Blocks are also used as the basis for elasticity on batch scheduling systems. Any scaling in/out must occur in units of blocks, as this is the most basic unit by which Parsl communicates with the scheduler. This abstraction can also be used to avoid limitations on the number of nodes that may be allocated concurrently or on the number of jobs that may be queued and/or running concurrently.

4.3 Executors

As illustrated in §2.1, Parsl's use cases vary widely in terms of their execution requirements. Individual Apps may run for milliseconds or days, and available parallelism can vary between none for sequential programs to millions for "pleasingly parallel" programs. Executors, as the name suggests, execute Apps on one or more target execution resources such as multi-core workstations, clouds, or supercomputers. As it appears infeasible to implement a single execution strategy that will meet so many diverse requirements on such varied platforms, Parsl provides a modular executor interface and a collection of executors that are tuned for common execution patterns. Figure 2 shows three such executors: high throughput, extreme scale, and low latency.

Parsl executors extend the Executor class offered by Python's `concurrent.futures` library, which allows us to use several existing solutions in the Python Standard Library (e.g., ThreadPoolExecutor) and from other packages such as IPyParallel [10]. Parsl extends the concurrent.futures executor interface to support additional capabilities such as automatic scaling of execution resources, monitoring, deferred initialization, and methods to set working directories.

All executors share a common execution kernel that is responsible for deserializing the task (i.e., the App and its input arguments) and executing the task in a sandboxed Python environment.

4.3.1 High Throughput Executor. The High Throughput Executor (HTEX) is a general-purpose executor, designed to enable high throughput execution of tasks using a pilot job model. It is engineered to support up to 2000 nodes, millions of sub-second tasks, and multi-day workflow campaigns, all while providing a high level of fault-tolerance. The HTEX architecture (see Figure 2a) has three major components: executor client, interchange, and managers.

HTEX managers (pilot agents) are deployed onto one or more nodes by the provider. Each manager is a multi-threaded agent responsible for a single node, initializing workers based on HTEX configuration (i.e., `workers_per_node`). It advertises available capacity, and receives batches of tasks from the interchange which are distributed to worker processes. Similarly, results are aggregated from workers and sent to the interchange in batches. The manager uses configurable *batching* and *prefetching* of tasks to minimize communication overheads.

The interchange is a hub to which the executor client and registered managers connect using ZeroMQ [32] queues. The interchange acts as a broker, matching available tasks to managers with

advertised capacity, while using a randomized selection method to ensure task distribution fairness.

Managers and the interchange exchange periodic heartbeat messages for fault tolerance purposes. If either party does not receive a message before a configurable threshold, the counterpart is assumed to be lost. Managers, upon losing contact with the interchange, exit immediately to avoid resource wastage. If the interchange detects the loss of a manager that had outstanding tasks, an exception is sent to the executor so that DFK can make appropriate decisions, such as scaling resources to match lost capacity and retrying failed tasks. HTEX also provides a separate command channel that can be used to perform administrative actions in a synchronous fashion. For example, the interchange can be asked for outstanding task information, to blacklist managers, or to shutdown the executor.

4.3.2 Extreme Scale Executor. The Extreme Scale Executor (EXEX) is designed to support the largest supercomputers—machines with thousands of nodes, hundreds of thousands of cores, and with specialized network architectures optimized for MPI. EXEX leverages MPI communication (using mpi4py [23]) to exploit the highly optimized network infrastructure to manage distributed execution.

The EXEX architecture (see Figure 2b) has three major components: EXEX executor client, interchange, and workers. EXEX is deployed as a multi-node batch job, that uses MPI for manager-worker communication and ZeroMQ for manager-interchange communication. EXEX uses a hierarchical task distribution model, where the managers communicate with the interchange on behalf of workers. Upon deployment, rank 0 of the MPI communicator takes the role of the *manager*, while all other ranks assume the role of workers.

Production runs over thousands of nodes are expensive, and unfortunately the likelihood of machine faults increase with scale. The primary drawback of using MPI as the communication fabric is that it reduces fault tolerance in the context of many-task applications [26]. Job and node failures can result in the loss of the entire MPI application. To alleviate these risks, we recommend that users break their allocation into several smaller MPI worker pools within a single scheduler job. EXEX is able to detect failures via the same heartbeat system described in §4.3.1.

4.3.3 Low Latency Executor. The Low Latency Executor (LLEX) is designed for use cases that require low latency function execution, but do not necessarily need high-throughput or fault-tolerance. Since the goal of LLEX is to minimize the round-trip-time for tasks, the execution model is designed to be as minimal as possible, thus sacrificing features such as reliability and automated resource provisioning for lower latency.

The LLEX architecture (see Figure 2c) has three major components: executor client, interchange, and workers. To execute tasks, the LLEX client forwards task information to the interchange, which in turn buffers and routes tasks to available workers. Results from workers are aggregated by the interchange and returned to the client. All network communication and routing of messages are handled by ZeroMQ. The interchange does not do any task tracking, and simply acts as a relay between clients and workers. This means that the routing logic is completely stateless and opaque to the interchange. As a result, while latency is reduced by avoiding task-tracking overhead on the interchange, failures such as worker loss cannot be detected by LLEX.
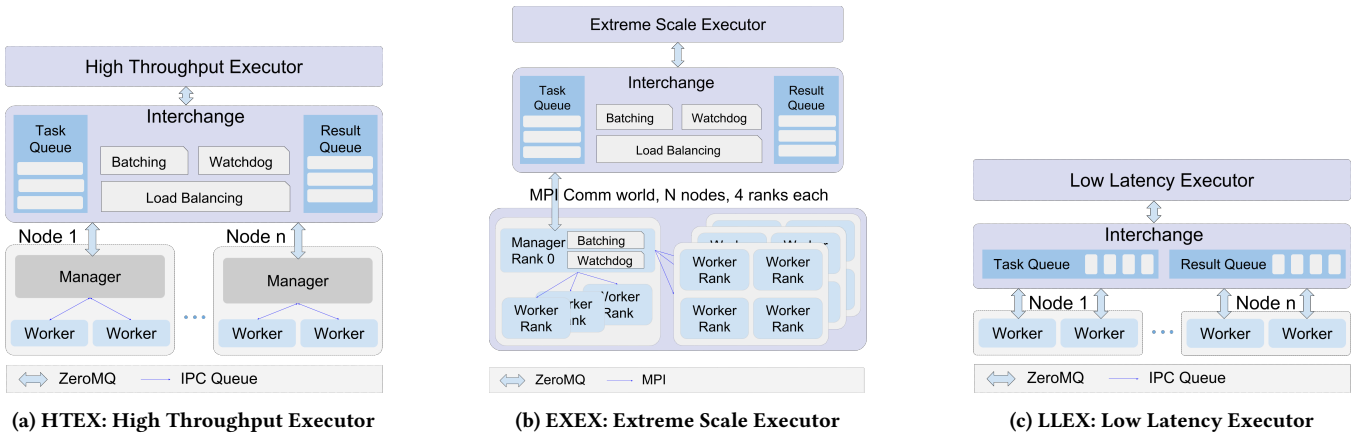
(a) HTEX: High Throughput Executor   (b) EXEX: Extreme Scale Executor   (c) LLEX: Low Latency Executor

Figure 2: Architecture of Parsl's high throughput, extreme scale, and low latency executors.

Unlike other executors, workers connect to the interchange directly. While this design requires a socket for each worker, message hops are reduced by one each way, reducing latency. Since tasks are short duration, reliable execution can be guaranteed with minimal cost, even on unreliable nodes, by *timed-retries* and *replication*

Finally, to meet high availability and latency requirements, LLEX assumes that it is operating on a fixed set of compute resources. Provisioning and relinquishing resources can take seconds to minutes on clouds and clusters, severely affecting task latencies.

4.4 Elasticity

Workload resource requirements often vary over time. For example, in the map-reduce paradigm the map phase may require more resources than the reduce phase. In general, reserving sufficient resources for the widest parallelism will result in underutilization during periods of lower load; conversely, reserving minimal resources for the thinnest parallelism will lead to optimal utilization but also extended execution time. Even simple bag-of-task applications may have tasks of different durations, leading to trailing tasks with a thin workload [15].

Parsl implements a cloud-like elasticity model in which resource blocks are provisioned/deprovisioned in response to workload pressure. Parsl provides an extensible *strategy* interface by which users can implement their own elasticity logic. By default, the elasticity strategy can be configured with a parallelism parameter that describes how aggressively the resources should grow and shrink in response to waiting tasks. Given the general nature of the implementation, Parsl can provide elastic execution on clouds, clusters, and supercomputers. Of course, in an HPC setting, elasticity may be complicated by queue delays.

4.5 Data management

Many use cases in §2.1 include Apps that pass files to/from one another. Hard coding file paths (either local or remote) breaks the execution location independence of a Parsl program. Parsl provides a *file* abstraction to allow file references between Apps. Parsl's data manager is responsible for transferring the file to where it is needed and for transparently translating the physical location as needed.

Parsl files can be defined either locally or using one of three data access protocols: HTTP, FTP, and Globus [20]. When a remote file is passed to/from an App, the Parsl data manager first inspects the file to see if it is available on the compute resource. If the file is not yet available, Parsl created a dynamic data dependency between the App(s) that require the file as input and a new (transparent) data transfer task. When the transfer is complete, the dependent App(s) are then able to execute. Parsl translates the file reference to a local path via which the App can access the file.

The data manager performs slightly different actions depending on the access protocol. In the case of HTTP and FTP files, the data manager creates a transfer task that is executed by the executor. That is, it is itself a task that is executed the same way as any other task. Globus does not require the task to be executed on the execution resource as it supports third party data transfer. When a Globus file is used, Parsl will introduce a transfer task into the graph; however, in this case the task is executed directly by the data manager which allows the deferment of resource provisioning until the data has been staged to the target resource.

4.6 Additional features

Parsl provides a range of other features that are desirable when developing parallel programs.

Authentication: Parsl integrates with Globus Auth [13] as a "native app." This allows users to authenticate in a program, either using interactive web-based login or cached access tokens. After authentication, access tokens are stored by Parsl, and these tokens are then used to securely access Globus Auth-enabled services (e.g., to transfer data or SSH to a compute resource).

Containers: Parsl allows workers to be launched inside a predefined container, allowing tasks to be executed in a customized environment. Parsl also allows containers to be used to execute tasks such that each invocation of a task will run a new container. Containers provide a popular way to package and distribute software and to deploy it in heterogeneous environments.

Monitoring: To enable both real-time and post-completion analysis and introspection of execution information, DFK logs execution metadata and task state transitions, and workers log task execution information, including resource usage. A modular DFK interface allows monitoring information to be stored in a SQL database, Elastic Search, or files. Logged data can be viewed via Parsl's web-based visualization interface.

Memoization: Parsl Apps can specify different levels of memoization to avoid repeated execution of the same App with the same input arguments. Memoization can be defined at both the program and individual App levels. This flexibility allows developers to select which Apps should be memoized, as memoization is rarely useful for non-deterministic apps. Parsl then maintains a cache of executed Apps, function body hash, and arguments.

5 EVALUATION

We evaluated the performance of Parsl with respect to latency, weak scaling, strong scaling, and elasticity using experiments conducted on two testbeds: Midway and Blue Waters.

We use the "broadwl" partition of the **Midway** campus cluster at the University of Chicago [11], each node of which has 28 Intel E5-2680v4 cores running at 2.4 GHz with 64 GB RAM, interconnected with Infiniband. The average network round trip time between two nodes was measured as 0.07 ms.

The **Blue Waters** supercomputer at the National Center for Supercomputing Applications [18] is a 13 petaFLOP Cray XE/XK hybrid system comprising 22 636 XE compute nodes (362 240 cores) and 4288 XK compute nodes (33 792 cores) with an additional 4228 Kepler Accelerators. We used the XE component. Each XE node has 16 AMD Interlargos cores (32 integer scheduling units) running at 2.3 GHz with 64 GB RAM, interconnected with the low-latency 3D Torus architecture. The average network round trip time between two nodes was measured as 0.04 ms.

We compare Parsl against three popular parallel computing libraries for Python. IPyParallel (IPP) [10] enables IPython to support parallel computing. We compare against an IPP-based executor for Parsl (since deprecated). FireWorks [33], a Python-based workflow management system, uses a centralized MongoDB-based LaunchPad to store tasks, and allows connected FireWorkers to query tasks from LaunchPad for execution. Dask distributed [9], a framework for parallel computing in Python, has a centralized scheduler that enables task submission, makes scheduling decisions, and executes tasks on connected workers.

5.1 Latency

We evaluated single task latency using the executors described in §4.3. The experiment was performed on two Midway nodes: one to run the Parsl program and the other to run a worker. To avoid including overhead, we first deployed the worker and waited for it to connect to Parsl. We then launched 1000 tasks sequentially, recording for each the time from submission until completion. For comparison, we also measured execution times for the same scenario on a single node using the ThreadPool executor, for IPP with Parsl on two nodes, and for Dask on two nodes. Figure 3 shows the distribution of task latencies in each case.

Our results show that LLEX (avg: 3.47 ms) is considerably faster and has lower latency variability than the other executors. LLEX is only approximately 2.43 ms slower than the local ThreadPool executor. As expected, HTEX (avg: 6.87 ms) and EXEX (avg: 9.83 ms) exhibit larger latencies due to the additional complexity of their respective executor architectures. The Parsl executors all have lower latencies than IPP (avg: 11.72 ms) and Dask (avg: 16.19 ms).
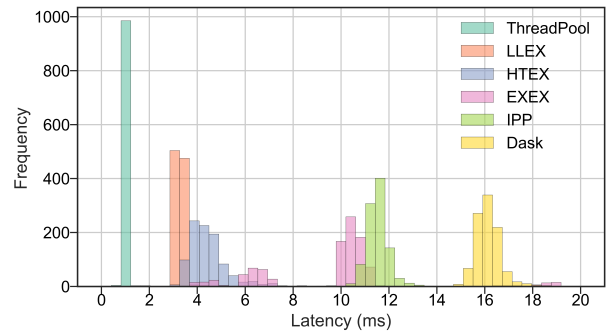

Figure 3: Distributions of task latencies when running 1000 tasks on Midway with different executors.

5.2 Scalability

We studied strong and weak scaling on Blue Waters. In strong scaling, the total problem size is fixed; in weak scaling, the problem size *per CPU core* is fixed. In both cases, we measure completion time as a function of number of CPU cores. An ideal framework should scale linearly, which for strong scaling means that speedup scales with the number of cores, and for weak scaling means that completion time remains constant as the number of cores increases.

To measure the strong and weak scaling of Parsl executors, we created Parsl programs to run tasks with different durations, ranging from a "no-op"—a Python function that exits immediately—to tasks that sleep for 10, 100, and 1000 ms. For each executor we deployed a worker per core on each node.

We compare the scalability of Parsl with IPP on Parsl, FireWorks, and Dask distributed. We launched the LaunchPad/scheduler of Fireworks and Dask distributed on one compute node, and the workers on the other compute nodes. For fair comparison, we deployed each worker process on one core and disabled caching (if available).

Strong scaling. We launched 50 000 independent tasks of different durations (no-op and 10, 100, 1000 ms) on an increasing number of workers and with different Parsl executors as well as IPP, and Dask distributed. For FireWorks we only launched 5000 tasks due to the limited allocation available to us on Blue Waters. Notice that the measurements for "no-op" tasks essentially reflect the overhead of the executor.

The top row of Figure 4 show the strong scaling results. HTEX provides the best performance in all cases, slightly exceeding what is possible with EXEX, while EXEX scales to significantly more workers than the other executors and frameworks. Encouragingly, both HTEX and EXEX remain nearly constant, indicating that they likely will continue to perform well at larger scales. In comparison with the other frameworks, FireWorks has the highest overhead even with only 5000 tasks: almost an order of magnitude greater than the other executors/frameworks. Both IPP and Dask distributed exhibit a similar trend of increasing overhead as the number of workers increases beyond 512. Dask distributed slightly outperforms HTEX and EXEX when there are fewer than 1024 workers.

Weak scaling. Here, we launched 10 tasks per worker, while increasing the number of workers. (We limited experiments to 10 tasks per worker, as on 3125 nodes, that already represents 3125 nodes × 32 workers/node × 10 tasks/worker, or 1M tasks.) The bottom row of Figure 4 shows our results. We observe that HTEX
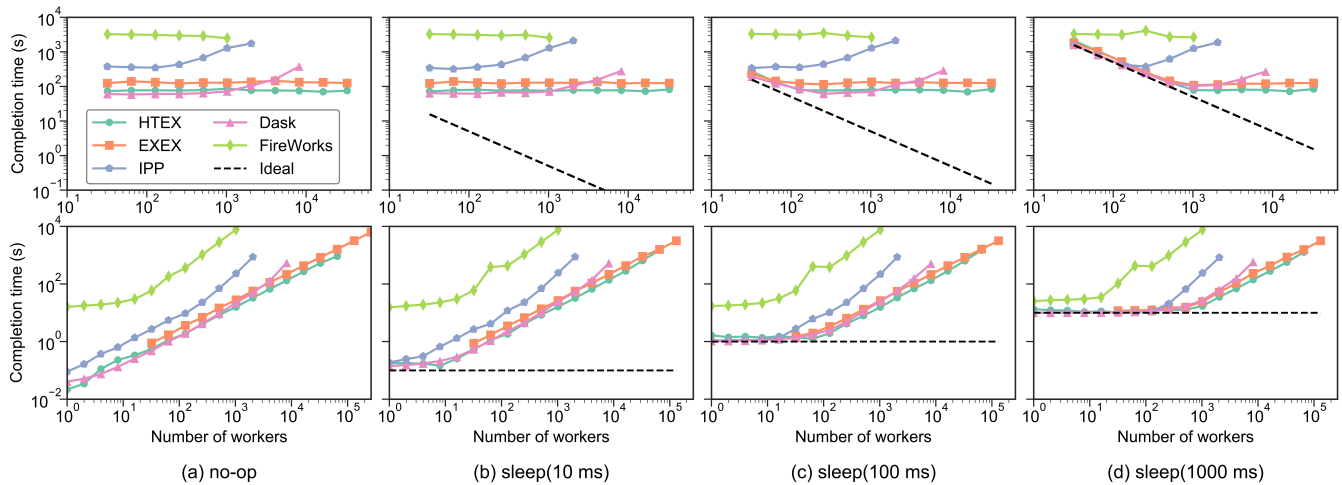
Figure 4: Top row: time to execute 50 000 tasks over all workers (strong scaling). Note: FireWorks results were obtained using 5000 tasks over all workers. Bottom row: time to execute 10 tasks per worker (weak scaling). For each row, plots are for (from left to right) tasks of 0, 10, 100, and 1000 ms. Legend is at top left.

and EXEX outperform other executors and frameworks with more than 4096 workers (128 nodes). While all frameworks exhibit similar trends, with completion time remaining close to constant initially and increasing rapidly as the number of workers increases, some executors/frameworks exhibit sublinear scaling more quickly than others. FireWorks scales sublinearly from around 32 workers, IPP at 256 workers (8 nodes), and Dask distributed, HTEX, and EXEX at 1024 workers (32 nodes).

Maximum number of workers. To further investigate scalability we now consider the maximum number of workers that can be connected without failure. To do so, we configured the executors on Blue Waters and continued to add workers until we observed errors. The maximum number of connected workers we observed are summarized in Table 2. We were able to launch 2048 workers on 64 nodes with IPP, 65 536 workers on 2048 nodes with HTEX, and 262 144 workers on 8192 nodes with EXEX. For HTEX and EXEX we were not able to force an error and were instead limited by the number of nodes we could provision in our allocation. Dask distributed scaled to 8192 workers on 256 nodes, after which we observed connection failures due to the fact that each worker must connect to the centralized scheduler, which can handle only a limited number of connections. FireWorks scaled to 1024 workers on 32 nodes, although at this point we observed slow performance and a variety of errors, such as time outs from its MongoDB server.

5.3 Throughput

We measured the maximum throughput of all the Parsl executors, as well as IPP with Parsl, Dask distributed, and FireWorks, on Midway. To do so, we ran 50 000 "no-op" tasks on a varying number of workers and recorded the completion times. The throughout is computed as the number of tasks divided by the completion time.

As shown in Table 2, IPP, HTEX, and EXEX achieved maximum throughputs of 330, 1181, and 1176 tasks/s, respectively. Dask distributed had the highest throughput, of 2617 tasks/s, likely as it is

optimized for short duration jobs on small clusters. FireWorks had the lowest throughout due to its slow centralized database.

Table 2: Capabilities and capacities of different Parsl executors and other parallel Python tools.

Framework	Maximum # of workers[†]	Maximum # of nodes[†]	Maximum tasks/second[‡]
Parsl-IPP	2048	64	330
Parsl-HTEX	65 536	2048[*]	1181
Parsl-EXEX	262 144	8192[*]	1176
FireWorks	1024	32	4
Dask distributed	8192	256	2617

[*] Limited by the the number of nodes we could allocate on Blue Waters during our experiments; this is not a scalability limit.

[†] These results are specific to Blue Waters, one core per worker, and using default configuration as in each framework's documentation.

[‡] The results in this column are collected on Midway.

5.4 Elasticity

We used the four-stage workflow shown in Figure 5 to study the efficacy of Parsl's elastic resource management. The first and third stages have the widest parallelism, with 20 tasks, while the second and fourth stage are reduce-like stages with a single task each. Every task is a sleep task and expends the capacity of a single worker for the specified duration (100 or 50 s). We executed this workflow on Midway with and without elasticity enabled and report the overall worker utilization of the acquired resources, calculated as the ratio of total wall clock time of tasks to that of the workers.

Figure 6 shows worker utilization when elasticity is enabled. Without elasticity, we observe average worker utilization of 68.15% and a makespan of 301 s. The worker utilization is poor during the reduce stages, wasting computing resources. In contrast, with elasticity enabled, average worker utilization is 84.28%, because Parsl can scale the number of blocks used dynamically, based on the workload. The makespan is slightly increased to 331 s. Overall,
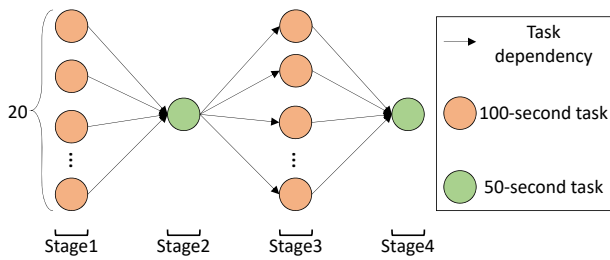
Figure 5: Workflow graph used in elasticity study.

in this example, utilization is increased by 23.6% at the expense of a 9.9% increase in makespan.
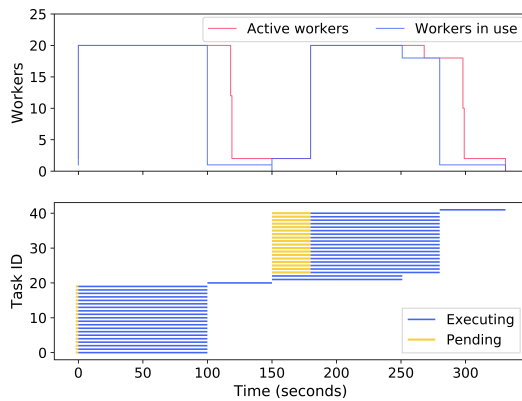

Figure 6: Utilization with elasticity. Above: Number of workers active and in use over time. Below: Task lifecycle, including both time waiting in a queue and time executing.

6 RELATED WORK

Analytics platforms. Hadoop [7] and Spark [8] are popular data-parallel systems for data analytics. Both follow the map-reduce model and are primarily designed for I/O-intensive applications, such as sorting, counting and aggregation. In contrast, Parsl implements a more general approach to parallelism, enabling various types of parallelism to be expressed in Python.

Scientific workflow engines. Many workflow systems—238 at the time of writing [2]—enable the orchestrated execution of multiple applications. Examples are Pegasus [24], Galaxy [31], Swift [39], NextFlow [25], FireWorks [33], Apache Airflow [6], and Luigi [4].

Pegasus and Galaxy implement a static DAG model in which users define a DAG, using an XML document or GUI, and subsequently execute that DAG. The Common Workflow Language (CWL) [1] attempts to standardize workflow descriptions using a YAML specification. These workflow systems take a different approach to addressing parallelism, requiring static, upfront definition of a workflow. Parsl instead augments Python, offering the full power of the Python programming language to create dynamic, parallel applications.

Swift and NextFlow rely on custom DSLs to express parallelism. While they provide for excellent performance; they have a steep

learning curve, and a limited set of programming constructs from which to create programs.

Python-based workflow systems such as FireWorks, Airflow, and Luigi enable the explicit description of dependency graphs in Python, rather than the ability to augment Python with parallelism. FireWorks focuses on fault-tolerance, rather than scalability and performance, relying on a persistent MongoDB to communicate task state. Airflow relies on a centralized scheduler that processes the task graph and manages execution on connected workers. Luigi, requires parallelism to be represented in classes, where a task describes its explicit input/output objects. When executed, Luigi builds a graph by introspecting the connected classes.

Parallel computing in Python. PyDFlow [14] was one of the first Python libraries to offer lazy evaluation in Python. Like Parsl, PyDFlow allows developers to wrap Python functions with decorators to denote lazy evaluation semantics. These functions produce I-vars [36] which, when accessed, produce a task graph that can be evaluated concurrently. Our experiences with PyDFlow have informed the development of Parsl.

The Dask [3] Python library supports parallel computing for data analytics. Unlike Parsl, Dask focuses on implementing parallel versions of common Python libraries, such as NumPy, Pandas, and Scikit-learn. Dask also offers low-level APIs for composing custom parallel systems, with constructs such as "delayed" for wrapping function calls, and futures for developing asynchronous programs. Dask distributed [9] extends Dask's execution model to support distributed execution on small clusters. It relies on a centralized scheduler that coordinates task submission and dynamic scheduling across multiple nodes. While Parsl and Dask share common features, Dask is primarily focused on data parallelism via high level libraries and on local or small-scale distributed execution environments.

Ray [35] is a distributed system designed to support training, serving, and simulation for reinforcement learning applications. Ray can execute millions of short-duration tasks. To achieve this performance it relies on a distributed scheduler and a distributed Redis-based fault-tolerant metadata store. The global control store maintains the entire state of the system, allowing the distributed scheduler to make rapid scheduling decisions based on global state. Ray implements a unified interface that enables expression of both actor and task-parallel abstractions for representing parallelism.

PyCOMPSs [38], a Python interface around the COMPSs system, is a framework for parallel computing in Python. As with Parsl, users decorate functions with constructs to aid workflow assembly and execution. COMPSs is responsible for interpreting the task graph and scheduling tasks to available resources.

While these frameworks implement a task graph model, Parsl focuses on a broader problem of enabling parallelism in Python. Parsl therefore tackles problems that range from many short tasks through to long tasks executing at extreme scale. The underlying model employed by Parsl could be used by many of these comparable frameworks to enable parallelism at a higher level.

Machine learning frameworks. TensorFlow [12] focuses on machine learning applications and can achieve high performance for linear algebra and other numerical computations. It represents computation as a dataflow graph, mapping each graph node to different machines or computational units (e.g., CPU and GPU).

LLEX for **interactive** computations on ≤**10** nodes.
HTEX for **batch** computations on ≤**1000** nodes. (*For good performance, task-duration / # nodes ≥ 0.01: e.g., on 10 nodes, tasks ≥ 0.1 s.*)
EXEX for **batch** computations on >**1000** nodes. (*For good performance, task durations ≥ 1 min.*)

Figure 7: Guidelines for selecting Parsl executors.

TensorFlow provides little support for more general parallelism, task composition, or other execution models.

7 SUMMARY

Parsl addresses two major trends in programming: the increasing use of high level languages, such as Python, to compose rather than write software; and the growing need for parallel computing in analysis and simulation. Parsl allows parallelism to be expressed via the use of simple decorators that enable safe, deterministic parallel programs; supports scalable execution from laptops to supercomputers; and provides a flexible architecture that can address the varied requirements of scientific analyses.

Our performance studies highlight the unique position in the parallel Python ecosystem that Parsl fills. Systems like FireWorks support use cases that require concurrent execution of few (<1000) long-running tasks (>100 s) [33]. Dask distributed can manage short tasks efficiently, but is designed for small-scale cluster deployments of fewer than 100 nodes. Parsl, and its flexible executor model, effectively fill the unmet needs of a variety of use cases, enabling efficient scalability up to ~8000 nodes (and likely more if allocations permit), execution overhead of less than 5 ms, and high-throughput execution of ~1200 tasks per second. We provide guidelines for selecting Parsl executors in Figure 7.

Our future work focuses on expanding Parsl capabilities. Having developed a flexible and general-purpose parallelism library, we next aim to investigate constructs for delivering parallelism such as maps and additional synchronization primitives such as barriers. We are particularly interested in supporting parallelism in higher-level libraries and domain-science libraries. We are also working to expand Parsl's data management capabilities, including by enabling direct data staging between nodes, ephemeral caching of data on nodes, and optional sandboxing environments.

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

ACKNOWLEDGMENT

REFERENCES

[1] Common Workflow Language Specifications, v1.0.2. https://www.commonwl.org/v1.0/. Accessed Apr 24, 2019.

[2] Computational Data Analysis Workflow Systems. https://s.apache.org/existing-workflow-systems. Accessed Apr 24, 2019.

[3] Dask. http://docs.dask.org/en/latest/. Accessed Apr 24, 2019.

[4] Luigi. https://github.com/spotify/luigi. Accessed Apr 24, 2019.

[5] Pickle. https://docs.python.org/3/library/pickle.html. Accessed Apr 24, 2019.

[6] Airflow. https://airflow.apache.org/. Accessed Apr 24, 2019.

[7] Apache Hadoop. https://hadoop.apache.org/. Accessed Apr 24, 2019.

[8] Apache Spark. https://spark.apache.org/. Accessed Apr 24, 2019.

[9] Dask distributed. http://distributed.dask.org/en/latest/. Accessed Apr 24, 2019.

[10] IPython.parallel. https://github.com/ipython/ipyparallel. Accessed Apr 24, 2019.

[11] Midway at University of Chicago Research Computing Center. https://rcc.uchicago.edu/docs/using-midway/index.html. Accessed Apr 24, 2019.

[12] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, et al. 2016. TensorFlow: A system for large-scale machine learning. In *12th USENIX Conf. on Operating Systems Design and Implementation*. 265–283.

[13] R. Anathankrishnan, K. Chard, I. Foster, M. Lidman, B. McCollam, S. Rosen, and S. Tuecke. 2016. Globus Auth: A research identity and access management platform. In *16th Intl Conf. on e-Science*. 203–212.

[14] T. Armstrong. 2011. *Integrating task parallelism into the Python programming language.* Master's thesis. University of Chicago.

[15] T. G. Armstrong, Z. Zhang, D. S. Katz, M. Wilde, and I. T. Foster. 2010. Scheduling many-task workloads on supercomputers: Dealing with trailing tasks. In *IEEE Workshop on Many-Task Computing on Grids and Supercomputers (MTAGS)*. 1–10.

[16] 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*.

[17] B. Blaiszik, K. Chard, J. Pruyne, R. Ananthakrishnan, S. Tuecke, and I. Foster. 2016. The Materials Data Facility: Data services to advance materials science research. *JOM* 68, 8 (2016), 2045–2052.

[18] 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.

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

[20] K. Chard, S. Tuecke, and I. Foster. 2014. Efficient and secure transfer, synchronization, and sharing of big data. *IEEE Cloud Computing* 1, 3 (2014), 46–55.

[21] 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*.

[22] R. Chard, R. Vescovi, M. Du, H. Li, K. Chard, S. Tuecke, N. Kasthuri, and I. Foster. 2018. High-throughput neuroanatomy and trigger-action programming: a case study in research automation. In *1st Intl. Work. on Autonomous Infra. for Sci.* ACM, 1:1–1:7.

[23] L. Dalcín, R. Paz, and M. Storti. 2005. MPI for Python. *J. Parallel and Distrib. Comput.* 65, 9 (2005), 1108–1115.

[24] E. Deelman, K. Vahi, G. Juve, M. Rynge, et al. 2015. Pegasus, a workflow management system for science automation. *Future Gen. Comp. Sys.* 46 (2015), 17–35.

[25] 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.

[26] M. Dorier, J. M. Wozniak, and R. Ross. 2017. Supporting task-level fault-tolerance in HPC workflows by launching MPI jobs inside MPI jobs. In *12th Workshop on Workflows in Support of Large-Scale Science*. 5:1–5:11.

[27] I. Foster, R. Olson, and S. Tuecke. 1992. Productive parallel programming: The PCN approach. *Scientific Programming* 1, 1 (1992), 51–66.

[28] I. Foster and S. Taylor. 1990. Strand: New concepts in parallel programming. *Prentice Hall* (1990).

[29] GFD-R-P.231 2016. *Distributed Resource Management Application API Version 2.2 (DRMAA)*. Recommendation. Open Grid Forum.

[30] GFD-R-P.90 2013. *A Simple API for Grid Applications (SAGA)*. Specification. Open Grid Forum.

[31] 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.

[32] P. Hintjens. 2013. *ZeroMQ: Messaging for Many Applications*. O'Reilly.

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

[34] M. M. McKerns, L. Strand, T. Sullivan, A. Fang, and M. A. Aivazis. 2012. Building a framework for predictive science. *arXiv preprint arXiv:1202.1056* (2012).

[35] P. Moritz, R. Nishihara, S. Wang, A. Tumanov, R. Liaw, E. Liang, M. Elibol, Z. Yang, W. Paul, et al. 2018. Ray: A distributed framework for emerging AI applications. In *13th USENIX Conf. on Operating Systems Design and Implementation*. 561–577.

[36] R. S. Nikhil. 1993. *An overview of the parallel language Id (a foundation for pH, a parallel dialect of Haskell)*. Technical Report. Digital Equipment Corporation, Cambridge Research Laboratory.

[37] J. J. Pitt. 2017. *Deciphering Cancer Development and Progression through Large-Scale Computational Analyses of Germline and Somatic Genomes*. Ph.D. Dissertation. University of Chicago.

[38] E. Tejedor, Y. Becerra, G. Alomar, A. Queralt, R. M. Badia, J. Torres, T. Cortes, and J. Labarta. 2017. PyCOMPSs: Parallel computational workflows in Python. *Intl Journal of High Performance Computing Applications* 31, 1 (2017), 66–82.

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