Parsl: Scalable Parallel Scripting in Python

Yadu Babuji*, Kyle Chard*, Ian Foster*, Daniel S. Katz§, Michael Wilde*, Anna Woodard*, 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*—Computational and data-driven research practices have significantly changed over the past decade to encompass new analysis models such as interactive and online computing. Science gateways are simultaneously evolving to support this transforming landscape with the aim to enable transparent, scalable execution of a variety of analyses. Science gateways often rely on workflow management systems to represent and execute analyses efficiently and reliably. However, integrating workflow systems in science gateways can be challenging, especially as analyses become more interactive and dynamic, requiring sophisticated orchestration and management of applications and data, and 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 wrapping either Python functions or calls to external applications. Parsl manages the execution of the script on clusters, clouds, grids, and other resources; orchestrates required data movement; and manages the execution of Python functions and external applications in parallel. The Parsl library can be easily integrated into Python-based gateways, allowing for simple management and scaling of workflows.**

I. Introduction

Data-driven research methodologies have had a disruptive impact on science, enabling new types of exploration and facilitating new discoveries [1]–[3]. Underlying these methodologies are new tools and technologies such as Jupyter notebooks for interactive analysis, scripting languages for flexible exploration, and a suite of libraries like Pandas and scikit-learn that facilitate cutting-edge analyses.

Science gateways [4] have long supported the varied needs of users, providing intuitive interfaces for end users to access both data and computing capabilities. Science gateway frameworks, such as Apache Airavata [5] and WS-PGRADE/gUSE [6], often rely on workflow frameworks to represent and execute analyses that benefit from extensibility, scalability, and robustness [7]. However, there are two significant challenges associated with current approaches: 1) many workflow engines are focused on many task applications rather than interactive, online, or machine learning analyses; and 2) workflow engines are not easily integrated into external services (e.g., gateways) due to issues such as language mismatch and the need for intermediate workflow representations.

Here we present Parsl, a Python parallel scripting library that supports the development and execution of asynchronous and implicitly parallel data-oriented workflows. Building on the model used by the Swift workflow language [8], Parsl brings parallel workflow capabilities to scripts, applications, and gateways implemented 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 arbitrary multicore processors, clusters, clouds, and supercomputers.

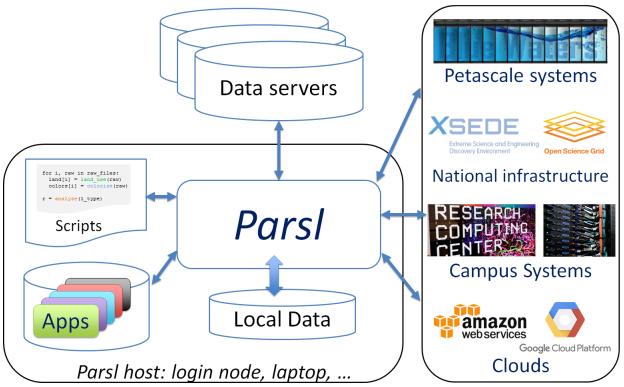


Fig. 1: Parsl environment.

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, referred to as *sites*. The execution fabric brings dependency awareness to Apps by introducing data *futures* as the inputs and outputs of Apps. Apps that use a data future as an input can be enqueued but will be blocked until that data future has been written. This feature allows Apps to execute in parallel whenever they do not share dependencies or their data dependencies have been resolved.

Fig. 1 depicts how Parsl interacts with its environment, including code, data, and resources. Parsl provides several advantages to science gateways: it allows a single script to be executed on *any* computing infrastructures from clouds to supercomputers; it provides fault tolerance, automated elasticity, and support for various execution models; it handles data management by staging local data through its secure message queue and by managing wide area transfers with Globus [9]; and it can be trivially integrated via its Python interface.

In this paper we describe Parsl, highlighting how it allows standard Python scripts and science gateways to be augmented to execute complex workflows and facilitate parallel execution. We describe Parsl's unique capabilities and present several example workflows that are common in science gateways from computational chemistry, materials science, and biology, to highlight the power of the approach.

II. Workflow models

Parsl is designed to support not only traditional many-task workflow models but also new analysis models that are and will be increasingly supported by science gateways (e.g., online and interactive computing). We briefly describe three such workflow models that can be supported by Parsl.

Workflows have long been applied to a range of many-task applications, for example protein-ligand docking for drug screening [10]. Here, workflows are used to orchestrate a series of external applications to be applied to a large set of input data. For example, in drug screening, dozens of proteins are evaluated against hundreds of thousands of drug candidates to identify the location and orientation of a ligand that binds to a protein receptor. The top candidates are then processed with detailed molecular dynamics simulations to identify the most likely combinations to be used for further experimentation. Gateways such as MoSGrid [11] and Galaxy [12] support such workflows.

Discovery science represents a new research methodology based on explorative, interactive analysis. The general model centers around analysis of large volumes of data with the aim to find unknown patterns. Notebook environments, such as Jupyter, provide an ideal interface in which researchers can discover and explore large data volumes using a variety of analytics approaches. Such methods are used in a wide range of studies from computing the stopping power of electrons through materials to measuring discursive influence across scholarship [13]. Gateways such as Cloud Kotta [14] and HubZero [15] expose Jupyter notebook interfaces for interactive computing.

Exploding data acquisition rates from scientific instruments, such as light sources, microscopes, and telescopes, necessitate rapid analysis to avoid data loss and enable online experiment steering. Real-time (or online) computing, such as that conducted at the Advanced Photon Source, allows for data streamed from beamline computers to be processed in real-time on a large cluster, with the aim to make real-time decisions during experiments [16].

III. Parsl model

The Parsl architecture is shown in Fig. 2. 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 users to annotate functions that will be run asynchronously based on data dependencies. The DFK provides a lightweight data management layer in which Python objects and files are staged to an execution site via a dedicated communication channel or Globus.

Dataflow Kernel: The DFK provides a single lightweight abstraction on top of 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.

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

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

Listing 1: Two examples of Parsl Apps.

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 1 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 checking status, blocking on completion, and retrieving results. Parsl leverages Python's `concurrent.futures` module for this purpose.

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

A. 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 and executors provide an abstraction layer for executing tasks.

Parsl's execution interface is called libsubmit [17]—a simple Python library that provides a common interface to execution resources. Libsubmit's interface defines operations such as submission, status, and job management. It currently supports a variety of providers including Amazon Web Services, Microsoft Azure, and Jetstream clouds as well as Cobalt, Slurm, Torque, GridEngine, and HTCondor Local Resource Managers (LRM). New execution providers can be easily added by implementing libsubmit's execution provider interface.

Depending on the the selected execution provider, there are a number of ways to submit workload to that resource. For

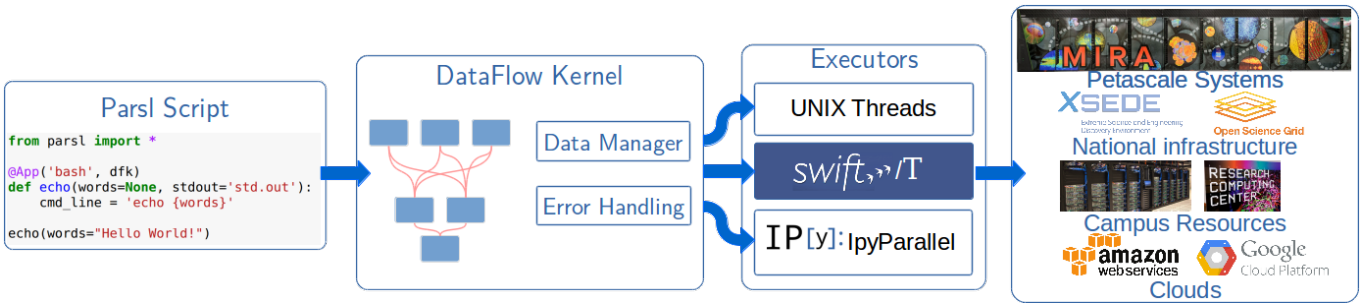


Fig. 2: Parsl architecture. The DataFlow Kernel maps scripts to Executors that support diverse computational platforms.

example, for local execution, threads can be used, while for a cluster, pilot jobs or specialized launchers can be used. Parsl supports these different methods via its *executor* interface. Parsl currently supports three executors:

- **ThreadPoolExecutor** for multi-thread execution on local resources.
- **IPyParallelExecutor** for both local and remote execution using a pilot job model. The IPythonParallel controller is deployed locally and IPythonParallel engines are deployed on execution nodes. IPythonParallel then manages the execution of tasks on connected engines.
- **Swift/TurbineExecutor** for extreme-scale execution using the Swift/T (Turbine) [18] model to enable distributed task execution across an MPI environment. This executor is typically used on supercomputers.

It is important to note that Parsl scripts are not tied to a specific executor or execution provider. Furthermore, a single Parsl script may leverage multiple executors and execution providers concurrently—a model we refer to as *multi site*. This allows Parsl developers to mix and match resources and execution models to meet their needs. For example, enabling a computational simulation to run on specialized HPC nodes, simple data manipulation tasks to be executed locally using threads, and visualizations to be rendered on GPU nodes.

B. Uniform execution model

Providing a uniform representation of heterogeneous resources is one of the most difficult challenges for parallel execution. Parsl provides an abstraction based on resource units called *blocks*. A block is a single unit of resources that is obtained from an execution provider. Within a block are a number of nodes. Parsl can then create *TaskBlocks* within and across (e.g., for MPI jobs) nodes. A TaskBlock is a virtual suballocation in which individual tasks can be launched. Figure 3 shows three different block configurations. The first configuration represents the most simple model in which a block is comprised of a single node with a single TaskBlock. The second configuration, with several TaskBlocks in a single node, is well suited for executing many, single threaded applications on a multicore node. The final configuration shows a block comprised of several nodes and offering several TaskBlocks. This configuration is generally used by MPI applications that span nodes. It requires specific MPI launchers supported by the target system such as aprun, srun, mpirun, and mpiexec.

C. Parallelism and elasticity

Rather than precompile a static representation of the entire workflow, Parsl implements a dynamic dependency graph in which the graph is constructed as tasks are enqueued. As the Parsl script executes the workflow, new tasks are added to a queue for execution, tasks are then executed asynchronously when their dependencies are met. Parsl uses the selected executor(s) to manage task execution on the execution provider(s).

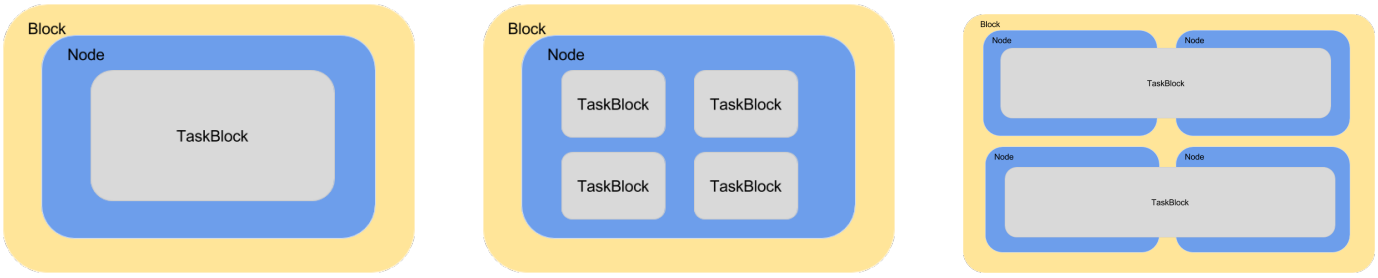
As Parsl manages a dynamic dependency graph it does not know the full "width" of a particular workflow a priori. Further, as a workflow executes, the needs of the tasks may change as too might the capacity available on execution providers. Thus, Parsl must elastically scale the resources it is using. To do so, it includes an extensible flow control system to monitor outstanding tasks and available compute capacity. This monitor, which can be extended or implemented by users, determines when to trigger scaling (in or out) events.

Parsl provides a simple user-managed model for controlling elasticity. It allows users to prescribe the minimum and maximum number of blocks to be used on a given execution provider and a parameter (p) to control the level of parallelism. Where parallelism is expressed as the ratio of TaskBlocks to active tasks. Each TaskBlock is capable of executing a single task at any given time. Therefore, a parallelism value of 1 represents aggressive scaling in which as many resources as possible will be used; parallelism close to 0 represents the opposite situation in which few resources (i.e., 1 TaskBlock) will be used.

D. Data management

Parsl is designed to enable implementation of dataflow patterns in which data passed between Apps manages the flow of execution. Dataflow programming models are popular as they can cleanly express, via implicit parallelism, the concurrency needed by many applications in a simple and intuitive way.

Parsl aims to abstract not only parallel execution but also execution location, which in turn requires data location abstraction. For Python Apps, Parsl uses a direct channel between the script and executors using Python object serialization. For



(a) A block comprised of a node with one TaskBlock.

(b) A block comprised of a node with several TaskBlocks.

(c) A block comprised of four nodes with two TaskBlocks.

Fig. 3: Parsl Block model showing several common block configurations.

files, Parsl implements a simple abstraction that can be used to reference data irrespective of its location. At present this model is limited to local and Globus [19] accessible files.

The Parsl file abstraction is used to pass location-independent references between Apps. It requires that the developer initially define a file's location (e.g., /local/path/file or globus://endpoint/file). The file may then be passed to each App and, when executed, Parsl will translate the location to a locally accessible file path. In the case of Globus, an explicit staging model is supported in which the developer must select the execution site to which the file should be transferred. Parsl uses the Globus SDK and its native App authentication model [20] to authenticate with the Globus service and securely move data between endpoints.

E. Caching

When developing a workflow, developers often execute the same workflow with incremental changes over and over, this scenario is especially prevalent in interactive computing workflows. Often large fragments of the workflow have not been changed yet are computed again, wasting valuable developer time and computation resources. Caching of Apps (often called memoization) solves this problem by saving results from Apps that have completed so that they can be re-used. Parsl's caching model stores App results in an index alongside the App function, input parameters, and hash of the function body. If caching is enabled, by an annotation on the App function or globally at the workflow level, the cache is interrogated before each App executes. Caching is supported for Python and Bash Apps. Users must explicitly enable caching to avoid issues with non-deterministic applications.

F. Checkpointing

Large scale workflows are prone to errors due to node failures, application or environment errors, and myriad other issues. Parsl provides fault tolerance via an incremental checkpointing model, where each checkpoint call saves all results that have been updated since the last checkpoint was created. When loading checkpoints, if entries with results from multiple functions (with identical hashes) are encountered, only the last entry read will be considered. Checkpoints are loaded from checkpoint files when the DataFlow Kernel is initialized and written out to checkpoint files when explicitly requested.

IV. Case Studies

We present three workflows implemented using Parsl to illustrate how it can satisfy the needs of different application domains. While these workflows have not yet been implemented in science gateways they represent use cases that would benefit from gateway models.

SwiftSeq [21] is a bioinformatics workflow that supports aligning and genotyping gene panels, exomes, and whole genomes. The Parsl-based 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 themselves 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 Apps, recover from errors, and execute on many computational resources.

Parsl has been used in computational chemistry to develop molecular dynamics workflows. In one example, PACKMOL [22] is used to assemble initial starting configurations of ionic liquid molecules with a protein (e.g., Trp-cage), before a GPU-accelerated version of Amber [23] is used to energy minimize, heat, equilibrate, and run production molecular dynamics simulations. 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.

In materials science, researchers have used Parsl to predict the electronic stopping power of materials. Stopping power is the predominant energy-loss mechanism for charged particles and is important for applications related to radiation protection. Historically, the stopping power for a material is computed using analytical models such as the Lindhard model or using Time-Dependent Density Functional Theory (TD-DFT). However, these methods are not suitable in all cases

and are computationally expensive. The Parsl-based workflow uses TD-DFT calculations of a proton passing through a material [24], transforms that data to a representation compatible with machine learning, and then executes a number of machine learning algorithms to learn a predictive model. It finally applies these models from various directions to calculate a three dimensional model of stopping power for a material. Parsl was used as it was able to trivially parallelize the existing Python codebase, support the composition of a sophisticated machine learning pipeline in a Jupyter notebook, and facilitate scalable execution of the pipeline from within the notebook on large-scale computing resources at the Argonne Leadership Computing Facility.

V. 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. Other systems include Pegasus [25] and Galaxy [12]. A weakness of these systems, however, is the need to develop a workflow representation in a separate representation (e.g., a graph) Parsl provides similar capabilities, directly in a programming language that is broadly adopted by scientific users and increasingly science gateways.

There are a number of Python-based workflow tools that better match common research environments, for example, Dask [26], Apache Airflow [27], Luigi [28], and FireWorks [29].

Dask is a parallel computing library designed for parallel analytics. It allows users to trivially migrate their single-node analyses to a parallel execution environment. Unlike Parsl, Dask scripts use Dask-specific functions in place of common libraries and programming constructs, for example using the Dask DataFrame in place of the Pandas DataFrame. Like Parsl, Dask decomposes a script into a dependent task graph that controls the execution of code blocks. Parsl focuses on a broader problem, including the ability to execute arbitrary applications on heterogeneous computing resources and providing support for managing data dependencies between these executions.

Apache Airflow is a workflow engine written in Python. Developers can express directed acyclic graphs of independent tasks. The Airflow scheduler is then responsible for executing the tasks on distributed workers according to their dependencies. Unlike Parsl's implicit workflow model, Airflow relies on users expressing their workflows as explicit tasks and with explicit relationships between those tasks. Thus, the job of the user is to essentially describe a task dependency graph in Python.

Luigi scripts are created by writing Python classes that extend the Luigi task model: developers implement functions that manage input and output data, the code that will be run, as well as the explicit dependencies on other tasks. Unlike Parsl, Luigi focuses on Python tasks rather than orchestrating execution of external applications. Further, Luigi offers a execution model that deploys workers on a single cluster; it is not designed to support multiple sites, provide elastic resource management, or handle wide area data staging.

FireWorks is a Python-based workflow engine designed for executing high-throughput workflows on supercomputers. Workflows are described in Python, JSON, or YAML and as a collection of tasks which are connected together into a "FireWork" for execution. The centralized server manages the workflow, using a MongoDB database to provide persistence and to support reliable execution on distributed resources. FireWorkers are deployed on compute resources to execute tasks, they connect to the centralized server to request tasks, execute them, and return results. Unlike Parsl, FireWorks focuses on the reliable execution of long running jobs and therefore may not be suitable for short running jobs or applications that demand a high submission rate.

VI. Summary

Parsl provides an easy-to-use model that can be easily integrated in science gateways to support the management and execution of workflows composed of Python functions and external applications. Science gateways benefit from the extensibility, scalability, and robustness of the Parsl model to manage execution of potentially complex workflows on arbitrary computational resources. Parsl is specifically designed to address new workflow modalities, such as interactive computing in Jupyter notebooks, and provides a seamless and transparent way to scale these analyses from within the notebook. Parsl abstracts the complexity of interacting with different resource fabrics and execution models. It instead supports the development of resource-independent Python scripts. It also includes a number of advanced capabilities such as automated elasticity, support for multi-site execution, fault tolerance, and automated direct and wide area data management.

Acknowledgment

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

References

[1] A. W. Toga, I. Foster, C. Kesselman, R. Madduri, K. Chard, E. W. Deutsch, N. D. Price, G. Glusman, B. D. Heavner, I. D. Dinov, J. Ames, J. Van Horn, R. Kramer, and L. Hood, "Big biomedical data as the key resource for discovery science," *Journal of the American Medical Informatics Association*, vol. 22, no. 6, pp. 1126–1131, 2015.

[2] N. P. Tatonetti, P. P. Ye, R. Daneshjou, and R. B. Altman, "Data-driven prediction of drug effects and interactions," *Science Translational Medicine*, vol. 4, no. 125, pp. 125ra31–125ra31, 2012.

[3] L. Ward and C. Wolverton, "Atomistic calculations and materials informatics: A review," *Current Opinion in Solid State and Materials Science*, vol. 21, no. 3, pp. 167 – 176, 2017.

[4] N. Wilkins-Diehr, "Special issue: science gateways - common community interfaces to grid resources," *Concurrency and Computation: Practice and Experience*, vol. 19, no. 6, pp. 743–749, 2007.

[5] S. Marru, L. Gunathilake, C. Herath, P. Tangchaisin, M. Pierce, C. Mattmann, R. Singh, T. Gunarathne, E. Chinthaka, R. Gardler, A. Slominski, A. Douma, S. Perera, and S. Weerawarana, "Apache Airavata: A framework for distributed applications and computational workflows," in *Proceedings of the 2011 ACM Workshop on Gateway Computing Environments*, 2011, pp. 21–28.

[6] P. Kacsuk, Z. Farkas, M. Kozlovszky, G. Hermann, A. Balasko, K. Karoczkai, and I. Marton, "WS-PGRADE/gUSE generic DCI gateway framework for a large variety of user communities," *Journal of Grid Computing*, vol. 10, no. 4, pp. 601–630, Dec 2012.

[7] T. Glatard, M. tienne Rousseau, S. Camarasu-Pop, R. Adalat, N. Beck, S. Das, R. F. da Silva, N. Khalili-Mahani, V. Korkhov, P.-O. Quirion, P. Rioux, S. D. Olabarriaga, P. Bellec, and A. C. Evans, "Software architectures to integrate workflow engines in science gateways," *Future Generation Computer Systems*, vol. 75, pp. 239 – 255, 2017.

[8] M. Wilde, M. Hategan, J. M. Wozniak, B. Clifford, D. S. Katz, and I. Foster, "Swift: A language for distributed parallel scripting," *Parallel Computing*, vol. 37, no. 9, pp. 633–652, Sep. 2011.

[9] K. Chard, S. Tuecke, and I. Foster, "Efficient and secure transfer, synchronization, and sharing of big data," *IEEE Cloud Computing*, vol. 1, no. 3, pp. 46–55, Sept 2014.

[10] A. N. Adhikari, J. Peng, M. Wilde, J. Xu, K. F. Freed, and T. R. Sosnick, "Modeling large regions in proteins: Applications to loops, termini, and folding," *Protein Science*, vol. 21, no. 1, pp. 107–121, 2012.

[11] J. Krüger, R. Grunzke, S. Gesing, S. Breuers, A. Brinkmann, L. de la Garza, O. Kohlbacher, M. Kruse, W. E. Nagel, L. Packschies, R. Mller-Pfefferkorn, P. Schäfer, C. Schärfe, T. Steinke, T. Schlemmer, K. D. Warzecha, A. Zink, and S. Herres-Pawlis, "The MoSGrid science gateway a complete solution for molecular simulations," *Journal of Chemical Theory and Computation*, vol. 10, no. 6, pp. 2232–2245, 2014.

[12] E. Afgan, D. Baker, M. van den Beek *et al.*, "The Galaxy platform for accessible, reproducible and collaborative biomedical analyses: 2016 update," *Nucleic Acids Res.*, vol. 44, no. W1, p. W3, 2016.

[13] A. Gerow, Y. Hu, J. Boyd-Graber, D. M. Blei, and J. A. Evans, "Measuring discursive influence across scholarship," *Proceedings of the National Academy of Sciences*, 2018.

[14] Y. N. Babuji, K. Chard, and E. Duede, "Enabling interactive analytics of secure data using cloud kotta," in *8th Workshop on Scientific Cloud Computing*, ser. ScienceCloud '17, 2017, pp. 9–15.

[15] M. McLennan and R. Kennell, "HUBzero: A platform for dissemination and collaboration in computational science and engineering," *IEEE Des. Test*, vol. 12, no. 2, pp. 48–53, Mar. 2010.

[16] T. Bicer, D. Gursoy, R. Kettimuthu, I. T. Foster, B. Ren, V. D. Andrede, and F. D. Carlo, "Real-time data analysis and autonomous steering of synchrotron light source experiments," in *13th IEEE International Conference on e-Science (e-Science)*, Oct 2017, pp. 59–68.

[17] "Libsubmit," https://github.com/Parsl/libsubmit.

[18] T. G. Armstrong, J. M. Wozniak, M. Wilde, and I. T. Foster, "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.

[19] K. Chard, S. Tuecke, and I. Foster, "Efficient and secure transfer, synchronization, and sharing of big data," *IEEE Cloud Computing*, vol. 1, no. 3, pp. 46–55, Sept 2014.

[20] S. Tuecke, R. Ananthakrishnan, K. Chard, M. Lidman, B. McCollam, S. Rosen, and I. Foster, "Globus Auth: A research identity and access management platform," in *12th IEEE International Conference on e-Science (e-Science)*, Oct 2016, pp. 203–212.

[21] J. Pitt, "Swiftseq," http://www.igsb.org/software/swiftseq.

[22] L. Martínez, R. Andrade, E. G. Birgin, and J. M. Martínez, "PACKMOL: A package for building initial configurations for molecular dynamics simulations," *J. Comp. Chemistry*, vol. 30, no. 13, pp. 2157–2164, 2009.

[23] D. A. Case, T. E. Cheatham, T. Darden, H. Gohlke, R. Luo, K. M. Merz *et al.*, "The Amber biomolecular simulation programs," *J. Comp. Chemistry*, vol. 26, no. 16, pp. 1668–1688, 2005.

[24] A. Schleife, Y. Kanai, and A. A. Correa, "Accurate atomistic first-principles calculations of electronic stopping," *Phys. Rev. B*, vol. 91, p. 014306, Jan 2015.

[25] E. Deelman, G. Singh, M.-H. Su, Y. Blythe, James Gil *et al.*, "Pegasus: A framework for mapping complex scientific workflows onto distributed systems," *Scientific Programming*, vol. 13, no. 3, pp. 219–237, 2005.

[26] M. Rocklin, "Dask: Parallel computation with blocked algorithms and task scheduling," in *Proc. 14th Python in Sci. Conf.*, 2015, pp. 130–136.

[27] Apache Airflow Project, "Apache Airflow," https://airflow.incubator.apache.org/.

[28] Spotify, "Luigi," https://github.com/spotify/luigi.

[29] A. Jain, S. P. Ong, W. Chen, B. Medasani, X. Qu, M. Kocher, M. Brafman, G. Petretto, G. Rignanese, G. Hautier, D. Gunter, and K. A. Persson, "Fireworks: a dynamic workflow system designed for highthroughput applications," *Concurrency and Computation: Practice and Experience*, vol. 27, no. 17, pp. 5037–5059, 2015.