Lightweight Function Monitors for Fine-Grained Management in Large Scale Python Applications

Tim Shaffer*, Zhuozhao Li†, Ben Tovar*, Yadu Babuji†,
TJ Dasso*, Zoe Surma*, Kyle Chard†‡, Ian Foster†‡, Douglas Thain*
*University of Notre Dame; †University of Chicago; ‡Argonne National Laboratory

***Abstract*—Python has become a widely used programming language for research, not only for small one-off analyses, but also for complex application pipelines running at supercomputer-scale. Modern parallel programming frameworks for Python present users with a more granular unit of management than traditional Unix processes and batch submissions: the Python function. We review the challenges involved in running native Python functions at scale, and present techniques for dynamically determining a minimal set of dependencies and for assembling a lightweight function monitor (LFM) that captures the software environment and manages resources at the granularity of single functions. We evaluate these techniques in a range of environments, from campus cluster to supercomputer, and show that our advanced dependency management planning and dynamic resource management methods provide superior performance and utilization relative to coarser-grained management approaches, achieving several-fold decrease in execution time for several large Python applications.**

I. Introduction

As researchers increasingly structure scientific applications based on fine-grained functions and reusable library components, they are essentially decomposing what were previously monolithic applications into collections of small and schedulable sub-components. Such decomposition is beneficial for many reasons, including modularity, extensibility, ease of development, and understanding. Furthermore, the resulting small tasks can be executed essentially *anywhere*. Thus, execution can be fluid: individual tasks may be routed to specialized computing resources, to data, or to available capacity. Additional resources may be provisioned rapidly to meet short-term fluctuations in concurrency. However, while many applications may benefit from such fluidity, we currently lack the resource management systems to efficiently and transparently map functions to available resources.

The need to support parallel and distributed programming in productive languages like Python has led to the development of libraries such as Dask [1], Parsl [2], and Ray [3]. In each, Python function invocations are dispatched to *workers* for execution. Workers, which are deployed at startup in one or more processes, are responsible for managing local resources on that physical node. Tasks are routed by the library to workers and are subject to the worker's environment and resource allocation. FaaS systems, such as Amazon Lambda [4] and Google Cloud Functions [5], enable registration of programming functions (including Python) alongside a list of static dependencies and coarse resource requirements (e.g., memory). These static dependencies are used to generate containers in which functions may be executed in isolation. While these systems enable users to program in terms of functions, their underlying resource management models are inherently process-based.

Most resource management systems for scientific computing—workflow managers, batch systems, file systems—focus on the Unix process as the primary unit of resource management, even when executing within containers. A process requires a certain number of cores and quantity of memory, and is associated with an executable program, input files that it will consume, and output files that it will produce. Consequently, resource managers focus on dispatching a process to a node, executing a process to completion, perhaps preempting or migrating a process to another node, and then accounting for the resources consumed by the whole process. From this perspective, the internal structure of a program is irrelevant and can be implemented in any language that can be compiled into or interpreted as a single executable. In many systems, the granularity has become even coarser, resulting in an entire virtual machine or container becoming the atomic unit of management, often to facilitate the capturing of complex software dependencies attached to a process.

We consider here the barriers to making individual function invocations the fundamental unit of resource management in a distributed system. A function invocation bears some similarity to a process invocation, in that it consists of a code fragment that requires some cores and memory to run. However, its data dependencies comprise formal arguments, return values, and perhaps global variables. Instead of a complete executable file, the invocation requires (at least) the code for the named function, and perhaps named import dependencies from standard (or custom) libraries. This code must all be deployed into a Python interpreter of the appropriate type and version. If many function invocations are to be run concurrently, then effective resource management requires that we measure the resource consumption of invocations accurately, and use that information to place invocations in the system.

These considerations apply to applications written in a variety of high-level languages used in scientific computing, such as Java, R, Julia, and Matlab. We focus here on Python, as it is a widely used high-level language for executing programs at scale and for composing programs from external libraries, scripts, and applications written in other languages. As is well noted, Python has a number of other technical limitations when

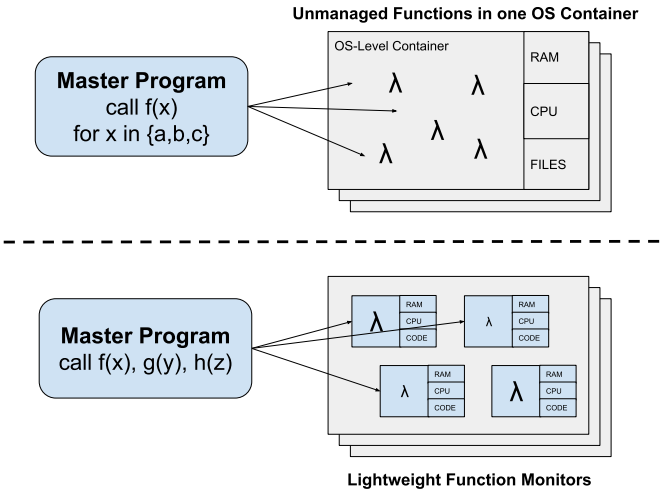

Fig. 1. Functions as first-class citizens. (Top) In a conventional FaaS application, concurrent function invocations are intermixed in a worker process managed by a container controlling resources and a file system at the operating system level. (Bottom) In our model, each function invocation has its resources and code independently measured and constrained in a LFM.

used in the high performance parallel and distributed systems available for scientific computing [6]. While Python has some native capability to use multiple threads on a single node, this capability is limited by the internal Global Interpreter Lock (GIL). Distributing Python programs to multiple nodes of a distributed or parallel system is made more difficult by the fact that a Python program is not a single executable. A scientific Python program often has dependencies on a complex set of Python libraries, as well as native C libraries that provide high performance implementations of algorithmic kernels. These dependencies present practical challenges to installation for end users, as well as to automated distribution of programs across multiple nodes.

The contributions of our work are: (1) An architecture that performs automated resource management for complex Python functions by extending and combining the Parsl parallel programming library with the Work Queue distributed execution framework. (2) Demonstrating how the direct deployment of Python function invocations into a native Unix environment results in serious inefficiencies. (3) Techniques for transparent dependency detection and distribution, such that a suitable execution environment for each function is loaded once and made available at each execution node, making use of shared file systems and local storage as needed. (4) Techniques for resource management in lightweight function monitors. As each function in a workflow is executed, we measure resources (cores, memory, I/O) consumed, and then group and efficiently pack functions into nodes. (5) Evaluation of these techniques on production applications that are representative of next-generation highly concurrent Python applications in physics and bioinformatics, as well as in a FaaS system.

II. Key Idea

Figure 1 illustrates the key idea of this paper. A conventional Function-as-a-Service (FaaS) application (top) is designed from the bottom up, by first defining a small number of well-characterized functions, and then designing an application that uses only those functions in a relatively uniform way across multiple data items. Each function's dependencies are determined by the user, packaged in advance, and deployed in a worker capable of executing only that fixed function. While the worker's overall resources may be limited by a container, individual function invocations are uncontrolled within the worker, assuming that they have relatively uniform behavior.

In contrast (bottom), we consider how to decompose rich Python applications that are composed of many existing functions with non-uniform and potentially complex behavior. Such functions may be long-running and have unpredictable resource consumption and complex software dependencies not easily determined by the end user. To make this possible, remote function invocation must become a first-class citizen such that resources (cores, memory, and software) are accurately requested, allocated, and monitored for each invocation. This allows the system to direct function invocations to the appropriate node(s) and to pack many function invocations into a node without exceeding the available resources.

To enable fine-grained management each invocation takes place in a *lightweight function monitor* (LFM), which provides the precise Python-level dependencies, monitors resource consumption, cancels functions that violate limits, and reports resource consumption. While similar in spirit to an OS-level container, the LFM uses Python-specific techniques to keep overhead low enough that containment can be applied to individual invocations.

III. Architecture

Figure 2 shows the general architecture of the applications considered in this paper, and the selection of specific technologies that implement that architecture. The application is divided between a set of coordinator processes that run on a suitable head node, and a set of workers that are deployed upon the nodes of the cluster. Each element of the architecture provides a step of the translation from high level user-facing code to individual tasks running concurrently on worker nodes.

Front-end. The end user accesses an application via a Python environment which allows for the entry of arbitrary Python code, and displays the output of that code. Note that, for consistency of experience, the user has no "side channel" into the cluster to configure or deploy any additional software or information relating to the underlying Unix cluster environment. All information flows through the Python interface, and the underlying components must infer any necessary configuration from that interface alone.

Parallel framework. The Python code written by the end user must be written in a framework that is capable of expressing a high degree of concurrency among tasks. The framework receives a specification of work to be done, divides it into suitably sized tasks, and then carries out the orderly

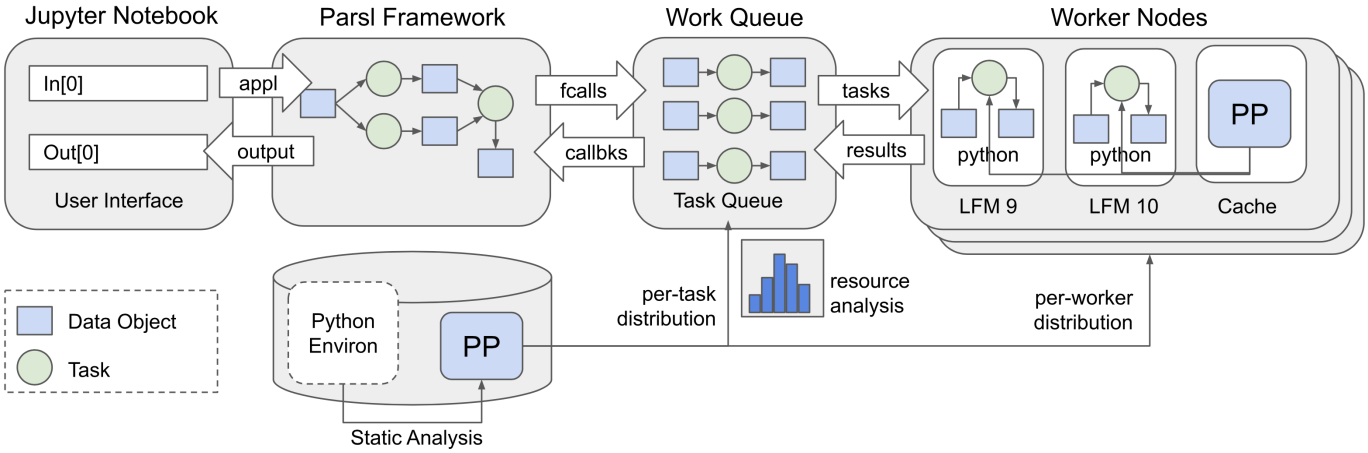

Fig. 2. Scalable Python application architecture. End users interact with an HPC facility via an interactive front-end environment, such as Jupyter, writing concurrent applications in Parsl. Parsl tracks the dependencies between function invocations and data items, passing discrete tasks to the Work Queue master process. Work Queue schedules tasks to worker nodes, along with their data dependencies. Each function invocation takes place in an LFM. Because users interact with the system entirely through the Python interface, all resource management is handled via the application without side-channel controls. For example, Python code dependencies are analyzed at the beginning of workflow execution and deployed automatically to the worker LFMs.

execution of the workflow by executing each task in order according to its dependencies, performing admission control to the underlying system, and handling the data transformation and management needed between each task. As each task becomes ready to execute, it is passed along to the task scheduler for placement in the cluster.

Task scheduler. The task scheduler maintains the list of tasks whose dependencies have already been satisfied and are ready for execution within the cluster. It is responsible for matching tasks to available workers by considering the resources needed by each task and available at each worker. The scheduler is also aware of the data items needed by each task, and may transfer (or direct the transfer of) those items.

Worker nodes. On each provisioned cluster node, a worker receives tasks sent by the scheduler, executes them as directed, and returns the results to the scheduler. A single task may consume only a fraction of available node resources, and so a worker must be able to execute multiple tasks simultaneously with appropriate isolation to prevent interference. Each task executes within an LFM, which tracks actual resource consumption, and reports observed behavior to the scheduler.

Cluster provisioning. The total number of worker nodes needed will certainly vary between applications, and may even vary at runtime if an application has highly dynamic needs. As a result, worker nodes must be provisioned at runtime by observing the workload (at the parallel framework) and submitting requests to start new workers, typically by submitting jobs to the native job scheduler on the cluster.

A. Implementation Technologies

We describe briefly the Parsl and Work Queue technologies that we use to build the applications considered in this paper. Users interact with these systems via a Python script or Jupyter notebook, which provides a structured interface for managing Python invocations and results without interacting with the Unix environment.

Applications are designed and executed using the **Parsl** [2] parallel programming library, which implements a dataflow-style programming model and a modular runtime model for executing concurrent tasks in Python. (Other popular parallel Python libraries could be used instead.) The Parsl model requires that developers annotate Python programs with function decorators representing which functions may be executed concurrently. Parsl supports annotation of Python functions and external applications invoked via the shell. Tasks execute asynchronously, with results returned as futures conforming to Python's `concurrent.futures` module. Evaluation of a future either yields the result or blocks until the result is available. Parsl establishes a dynamic dependency graph (as a DAG) as a program is executed by tracking the futures passed between functions. The Parsl library manages the DAG and determines which tasks can be executed. These tasks are passed to a pre-configured task scheduler.

For task scheduling, we use **Work Queue** [7], a master-worker framework for building large applications spanning thousands of nodes drawn from clusters, clouds, or grids. Work Queue accepts tasks in the form of Unix command lines, with explicit input and output files used to construct the namespace of the task. Here we developed a new Parsl-Work Queue executor module to map pending Python functions to Work Queue tasks, such that each task consists of an invocation of the appropriate Python interpreter with function inputs "pickled" (serialized) into transferable files (i.e., executed in an LFM). As tasks complete, their outputs are pickled for transfer back to the scheduler, where they are converted back into native Python data structures, and used to satisfy pending futures. Frequently used files are cached at the worker to facilitate re-use, and the master prefers to schedule tasks where needed data is cached.

B. Applications Used for Evaluation

For our evaluation, we selected several Parsl-based scientific applications that are currently in use across a variety of computing environments, and that significantly differ in their organization and behavior (see Figure 3).

HEP. Traditional High Energy Physics (HEP) analyses rely on successive processing steps to reduce an initial dataset (typically, 100s of PB) to a size that permits real-time analysis. To improve the flexibility of analysis and enable real-time analysis, researchers are adapting the traditional model to support native operations on hierarchically nested, columnar data. Coffea [8] is a new analysis framework which provides tools for histogramming, plotting, and performing data transformations and corrections. Coffea's use of columnar analysis is markedly different from the traditional HEP paradigm of event loop analysis, which operates row-by-row instead of column-by-column within a given dataset. Each row of input data corresponds to a single particle collision event, consisting of multiple properties (e.g. number of electrons or muons). A column therefore contains the values produced by *all* events for a specific property. By using columnar analysis, summary statistics for each field can be calculated individually and greater parallel processing of large datasets is possible.

Drug Screening Pipeline. The molecular search space for potential drug candidates that may be used to inhibit diseases is enormous. Computational approaches can provide a first level screen of potential candidates before being synthesized and used in lab and clinical trials. For some time, the computational modeling process has relied on expensive molecular dynamics simulations to predict the likelihood of protein-protein docking. To further expedite screening, researchers use machine learning models to predict candidates that are likely to produce high docking scores. An example workflow that is used to screen billions of small molecules for potential candidates to inhibit docking to COVID-19 proteins [9] proceeds as follows. The workflow first converts the SMILEs representation [10] for each molecule to a canonical form. It then creates three features for each: 1) a molecular descriptor; 2) a molecular fingerprint that encode the structure of molecules; and 3) a 2D image of the molecular structure. These features are used as input to two TensorFlow-based machine learning models that are trained on protein docking simulations to predict docking scores for candidate molecules. This process combines several independent functions and consumes millions of core hours to process billions of molecules.

Genomic Analysis. The GDC DNA-Seq analysis pipeline [11], which aims to identify variants between normal and tumor genomes, can take days or weeks to run on a single processor. The pipeline includes genome alignment, alignment co-cleaning, variant calling, variant annotation, and mutation aggregation tasks, each of which may rely on specific biology tools and have resource requirements that are hard to predict before the pipeline executes. For example, the pipeline uses a tool called Ensembl Variant Effect Predictor (VEP) [12] to annotate the effect of variants. However, VEP resource usage (e.g., memory usage, number of cores) depends on the number of variants in the data.

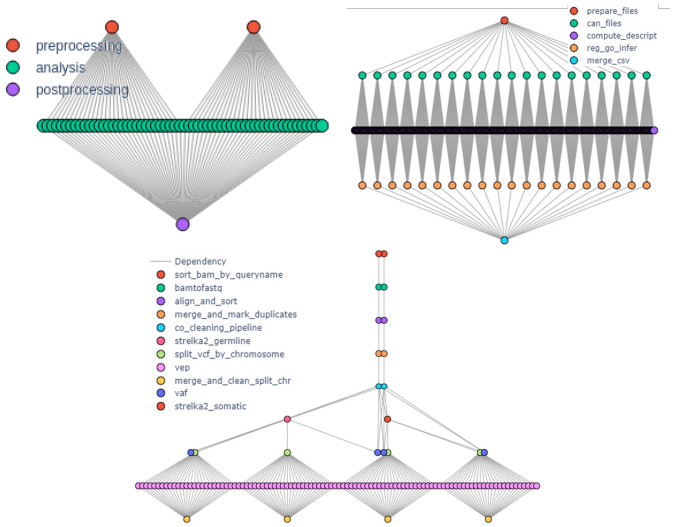

Fig. 3. Dependency graphs for HEP (top left), Drug Screening Pipeline (top right), and Genomic Analysis (bottom). Note that funcX operates under the Bag of Tasks model, so no workflow is given.

IV. Challenges

We sketch the problems that can occur when users deploy Python applications in the conventional way.

A typical installation places software in the user's home directory, relying on a shared file system to ensure that a common Python environment is accessible across nodes. Parallel execution is accomplished by submitting work to a batch scheduler. Since general-purpose batch systems must be language and application agnostic, units of work are necessarily coarse; depending on the site and queue configuration, jobs often occupy entire nodes. It is left to the application developer to determine how nodes and work should be subdivided to accommodate the granularity of the batch job.

While the above process may work in some situations, it is laborious, error-prone, unlikely to scale, and is inefficient. Most importantly, it is not designed for granular functions. To further highlight impediments, we describe open challenges and partial solutions for typical problems when deploying Python at scale. We also summarize the implications of these challenges.

Representing granular parallelism. Python is infamous for its concurrency limitations, primarily due to the GIL. However, there are many Python libraries that can be used to parallelize applications across cores and nodes. For example, the `multiprocessing` module in Python's standard library enables applications to make use of multiple cores and several Python libraries such as Dask, Ray, and Parsl enable scalable execution in distributed systems. Unfortunately, existing methods rely on access to shared file systems and homogeneous Python environments across nodes; furthermore, they do not typically provide methods to efficiently subdivide nodes to meet workload requirements. *Implication:* Current frameworks

are limited in terms of coarse resource management and the environments in which they can be deployed.

Managing Python environments. Python programs require access to specific Python versions, external packages, and ad hoc code accessible on the local file system. Python package managers provide mechanisms to assemble and clone such environments locally, and to import modules from remote repositories; however, in HPC environments these are not broadly supported, nor are they always efficient due to their reliance on shared file systems. *Implication:* There is significant manual overhead to assemble environments, make environments accessible, and to diagnose environment differences.

Determining environment requirements. While Python package managers support deterministic assembly of environments based on a list of requirements, they make no effort to optimize the environment to include only those packages that are strictly necessary for individual function execution. *Implication:* Execution environments are typically much larger than they need be and sometimes miss packages that are imported locally via PYTHONPATH and relative locations.

Distributing and configuring worker environments. Parallel Python libraries are notoriously fragile, requiring that master and worker software environments are nearly identical, from the interpreter version to each individual library. This often manifests as a "chicken and egg" problem and therefore requires that users manually deploy worker environments such that the parallel execution fabric can be deployed. *Implication:* Applications fail with little explanation, and so frameworks must take active steps to distribute consistent environments.

Minimizing load time for dependencies. Python environments can be large, exceeding several gigabytes and thousands of files for common scientific libraries. As a result, the time needed to load an environment can be significant and in some cases may be greater than the run time of a particular function. *Implication:* Overheads significantly reduce performance and may adversely affect shared file system performance.

Fine grain resource management. Existing systems focus on process level management and typically require that users manually determine resource needs—a task that is well-known to be inaccurate even at the granularity of Unix processes [13]. Providing efficient execution of fine grain functions requires that nodes be subdivided at equally granular levels, for example, in terms of cores or megabytes of memory. *Implication:* Resources are used inefficiently due to lack of function-level resource management and estimation.

We address these issues by defining an LFM, focusing on distributing Python environments (§V) and fine-grain function-level resource management (§VI).

V. Distributing Python Environments

We describe how we identify Python dependencies, create portable environments, and distribute environments to workers.

A. Observations

To illustrate difficulties that can arise when creating and using Python environments, we review application requirements and experiment with Python load times on HPC systems.

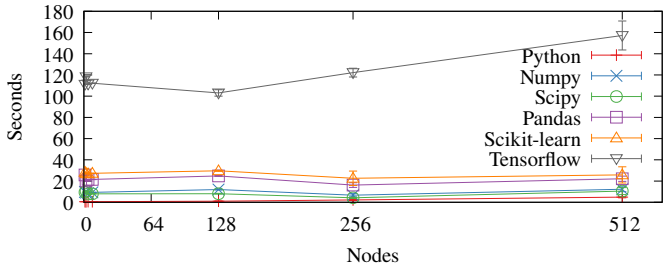

Fig. 4. Shared file system import time for libraries on Theta

1) Import Overhead: Figure 4 shows the time to import Python and various Python libraries on Argonne's Theta supercomputer as we scale from 64 to 32,768 cores (1 to 512 nodes). On each core we run a Python script that loads Python and imports a single module. We measure the time to run the script and plot the average loading time. We see constant performance for smaller modules, likely due to the fact that there is minimal data to load and little file system contention. For the larger TensorFlow, load time increases with the number of nodes, leading to significant wasted time. Figure 5 shows the cumulative time spent importing a single library on different systems when loaded from the shared file system (direct access) vs. unpacked on ephemeral disks from Conda (local unpack). On many nodes, cumulative time is many hours. Every function call on every node contributes to this overhead. Thus over the course of an application, we expect to pay this penalty many times over. Executing many short-lived functions or increasing the size of the worker pool further compounds this cost. Prior work has shown that this library loading overhead is primarily the result of heavy concurrent metadata load on the shared file system [14, 15].

B. Static Dependency Analysis

Irrespective of how Python environments are packaged and distributed, we first need to determine what packages a function needs to execute. In some cases a README enumerates installation steps or a Python pip requirements file lists packaged dependencies; however, these methods are error prone and often incomplete. A conservative approach would be simply to copy the user's entire Python environment. In practice, this is often ill-advised; users install many packages in their personal environment that are not needed for every application, let alone function. To produce a truly transparent solution, automated methods are needed to determine dependencies. We build upon prior work in static program analysis [16–18] to identify function dependencies.

For this work we developed a tool that can introspect a fragment of Python code (e.g., a function) and determine the modules needed to execute it. We use static analysis as it provides a simple way to trace import statements through packages and does not require executing the code. This approach is not foolproof in the general case. For example, it is possible to import Python modules via function call at runtime rather than by using static `import` statements. Parsl, however, requires that any libraries used by a function must

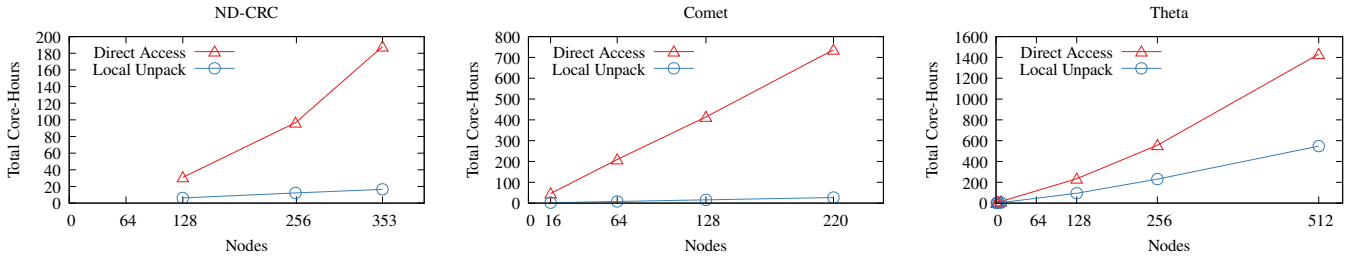

Fig. 5. Total time across pools of nodes to import TensorFlow. This overhead is incurred for each parallel function invocation that depends on TensorFlow. For Direct Access, TensorFlow was directly imported from an environment on the shared filesystem. For Local Unpack, this same environment was first packaged, then at runtime unpacked onto node-local storage.

be imported statically at the beginning of the function body, so static analysis is sufficient. For each Parsl function to be executed on remote nodes, the analysis tool uses Python's built-in parser and AST manipulation facilities to scan for `import` statements (or variations thereof). Each function can be analyzed in isolation from other functions and the rest of the program, greatly simplifying analysis and allowing for minimal dependency sets.

Once we determine which libraries a function uses, we query the user's current Python environment to identify the installed version of each imported package and add it to a list of dependencies. It is not necessary to include the full dependency tree, as Python package managers provide robust solvers for collecting dependencies recursively. We have integrated our static analysis tool with Parsl to parse the requirements of any Parsl functions and emit a list of requirements.

C. Packaging Python Environments

To explore different methods for managing Python environments we measured the time to load Python in Conda and container environments on several HPC systems. Specifically, we measured the time to run a simple "Hello World" function in a standard Python 3 environment. We compared performance in Singularity, Shifter, and Docker containers on the Theta supercomputer at Argonne, NERSC's Cori supercomputer, and AWS EC2, respectively. Table I shows our results. Conda is significantly faster than containers for packaging Python environments. To activate an environment, Conda needs only to make changes to environment variables for the running process. The container solutions, on the other hand, perform a variety of additional operations: creating kernel namespaces, mounting disk images, and preparing IO/resource controllers.

TABLE I
CONDA AND CONTAINER INSTANTIATION TIME FOR DIFFERENT CONTAINER TECHNOLOGIES ON DIFFERENT RESOURCES.

System	Conda/Container	Min (s)	Max (s)	Mean (s)
Theta	Conda	0.026	0.073	0.042
Theta	Singularity	2.327	3.50	2.628
Cori	Shifter	7.25	31.26	8.49
EC2	Docker	1.74	1.88	1.79

While our approaches could be applied to different virtual environments or containers, we focus on Conda for several reasons. Since Conda is an unprivileged user-level tool, it can be installed in any environment without administrator privileges. It is independent of any container technology, so the same process can be used whether a site has Docker, Shifter, or no container support at all. Finally, Conda is widely supported, contains a large number of Python packages, and automatically handles dependencies and installation steps.

D. Distributing Worker Environments

We now turn to the problem of distributing Python environments to worker nodes. We describe below several possible methods that may be advantageous under different conditions.

Loading directly from shared file system. The simplest way to access Python environments is to use a shared file system; however, increasing environment size and number of concurrent workers will lead to poor performance.

Dynamically configuring worker environments. The dependency list serves as a recipe for creating an environment on-demand. We can therefore transfer the list to the worker and use Conda to create the environment. This approach does not require a shared file system; however, it relies on outbound network access on the worker node to download required packages. Further, the installation process can be slow and concurrent downloads may result in network contention.

Transferring packed environments. An alternative approach is to create the Python environment on the master, package it, and send it to worker nodes. Each node can then unpack the environment locally and handle all task activity using fast local storage. Here we use `conda-pack` [19] to capture the environment in a tarball based on the dependency list. As the package is simply a file, we can transfer it to worker nodes using the shared file system, network, or burst buffers without producing the large metadata load associated with direct access. To use one of these tarballs on a worker node, we first extract the contents of the archive, then reconfigure the package for its new LFM.

E. Evaluation

We evaluate our environment packaging and distribution approaches on four HPC systems as outlined in Table III.

1) Packaging Costs: Table II shows for various packages the cost to analyze, create, and run the package via the shared file system; the package size; and the total number of packages

TABLE II
EXAMPLE PACKAGING COSTS (TIME TO ANALYZE DEPENDENCIES, CREATE PACKAGE, RUN PACKAGE), PACKAGE SIZE, NUMBER OF DEPENDENCIES.

	Time in seconds			Size	Dep
	Analyze	Create	Run	MB	Count
Python	1.20	23.0	2.7	82	21
NumPy	1.52	28.9	3.7	104	27
TensorFlow	3.97	60.9	10.7	259	58
Hyperopt	2.90	37.8	4.8	132	36
MXNet	2.89	72.8	15.6	348	93
Pandas	1.97	35.7	4.4	118	31
Spacy	2.25	39.6	5.2	127	55
Drug Screening Pipeline	18.50	200.0	30.0	833	163
HEP	27.00	149.0	8.8	204	92
Genomic Analysis	147.00	220.0	22.0	760	107

TABLE III
RESOURCES AT OUR TEST COMPUTING ENVIRONMENTS.

Site	Node CPU	Node Memory	Node Storage	Shared FS
ND-CRC	24 core Xeon	256 GB	SSD	Panasas
Comet	24 core Xeon	128 GB	SSD+Disk	Lustre
Cori	68 core Xeon Phi	96 GB	SSD+Disk	Lustre
Theta	64 core Xeon Phi	192 GB	SSD+Disk	Lustre

that are transitively required by the package (dependency count). We provide these data for the Python interpreter alone (which itself depends on several non-Python packages provided via Conda); the widely-used NumPy package; five packages selected from the Python Package Index (PyPI) based on the "SCIENTIFIC/ENGINEERING" label and with high download counts; and our three applications.

The TensorFlow and MXNet machine learning packages, in particular, depend on numerous other packages, increasing both the cost of all three package operations (analyze, create, run), and overall package size. So too do the three scientific applications; in their case, the large number of dependencies is due to their use of non-Python sub-components that are invoked by the Python functions, leading to a need for other components such as Java runtimes and Perl modules. However, while these dependencies increase costs associated with automatic package management for these applications, the costs must be balanced against the considerable effort that would be required to manually prepare the complex execution environments with many language runtimes and required dependencies.

2) Distributing worker environments: We measured the cost of distributing environments containing common packages (NumPy, SciPy, Scikit-learn, Pandas, TensorFlow) on several sites when using either the shared file system directly or by transferring the environment to node-local storage. We conducted these experiments by importing libraries concurrently across an increasing number of nodes, simulating the behavior of batch job/tasks starting together and executing the same code across many nodes. Figure 5 shows the import cost using either direct shared filesystem access or unpacking the software environment to node-local storage. We show results for TensorFlow only as it is more representative of real applications with many dependencies of varying size and complexity. Note that all three sites show an increase in overhead as the number of nodes increases, irrespective of the distribution method. In each case, transferring the environment using the shared file system and unpacking it locally significantly outperforms the use of the shared file system directly.

VI. ADAPTING TO CLUSTER RESOURCES

We now look at LFM resource management on clusters.

A. Observations

In traditional distributed applications, the relationship between units of work and the execution context is simple: users submit a command to the batch system, and the command process is then launched on a node and run until completion. When dealing with functions as first-class citizens in large applications, the relationship becomes more complicated. Long batch system latencies (days for large jobs on heavily used systems) and coarse time division make it infeasible to run short or interactive functions directly on a batch system. Likewise, since Python has strong support for running computations on multiple cores, choosing to map each function to a single core is ill-advised as native Python functions have no way of knowing what share of the node they are expected to use. Currently, Parsl configuration is done statically across an entire pool of workers, with each function assumed to have identical requirements. In practice, resource requirements can vary widely between different functions.

Even if expected resource requirements are carefully matched to the static configuration of nodes, minor changes in the application can cause unexpected resource conflicts later. NumPy is an excellent example of this issue, as a popular library for performing efficient computations on matrices. The default mode of operation for NumPy's matrix operations is single-threaded. When built against another library, BLAS, however, some of NumPy's operations switch to multicore implementations. There is no user-visible interface to control which implementation is used. The same application code, calling the same package and version, may nonetheless see striking differences in resource utilization depending on the build environment used. Even careful per-function resource configuration would not be sufficient in this case.

B. Approaches

Efficient execution of individual functions requires several changes in approach compared to traditional batch submission. Pilot job systems, such as Work Queue, offer a solution: long-lived agent processes are submitted that, upon execution, connect back to a master program. This approach allows us to maintain a pool of nodes that can execute lower-latency tasks.

As modern nodes have many cores (at least 20 in our test environments) there is potential for significant slowdown when assigning functions to whole nodes. Work Queue provides the ability to dynamically pack tasks onto available worker nodes if tasks are labelled with resource requirements. Thus we need

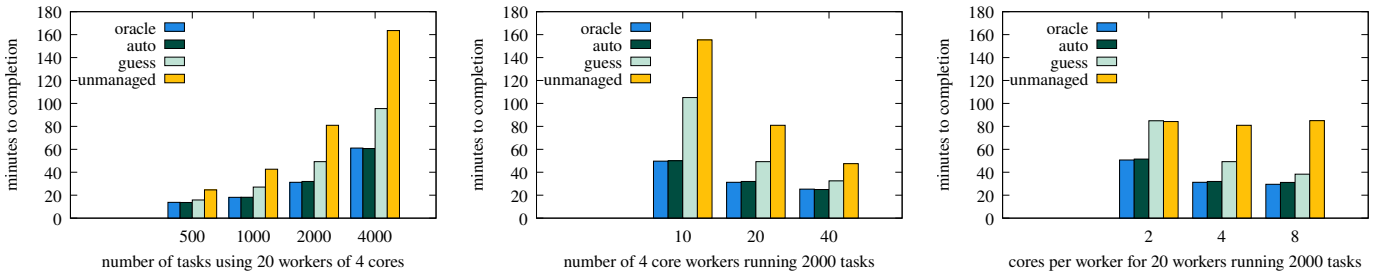

Fig. 6. HEP. **L**: 500–4000 tasks on 20 × 4-core workers. **C**: 10–40 × 4-core workers, 2000 tasks. **R**: 2–8 cores per worker, 20 workers, 2000 tasks.

methods for automated discovery of Python function resource usage, and for using that information for function packing.

1) Function-level resource monitoring: Aside from the lack of a language-level mechanism to define the resource properties of individual Python functions, it can be burdensome or even impossible for users to provide this information statically. Monitoring the resources used by a function invocation, such as cores, memory and disk, presents an interesting challenge for tasks defined as native Python functions. Unix machinery for measuring resource use, such as the `/proc` file system and `getrusage`, is designed to measure processes. A simple solution is to run a Python interpreter per task and measure the resources used by the processes spawned by the interpreter. However, the overhead of launching an interpreter per task quickly becomes prohibitive for short-running tasks. Further, a single interpreter cannot execute more than one task.

Instead, for each task we create a new process that we can measure with the `psutils` package or the Work Queue Resource Monitor [20]. The new process is initially a copy of the original Python interpreter and thus has access to the memory state before its creation. However, as Unix processes execute in their own copy-on-write memory space, any changes, including any task results, are lost when the new process terminates. To overcome this problem, we establish, before the new process is created, a queue for communicating results between the original Python interpreter process and the new task process. The task's function is wrapped such that its results (or its stack traceback in case of an exception) are returned via the queue to the original process. This setup also allows us to implement enforcement of resource limits. If during monitoring a tasks uses more resources than a specified limit, then its process can be terminated without terminating the original Python interpreter.

The resource measurement is done using two techniques: polling and process creation/exit events. With polling, at given intervals we read process information from `/proc/PID/`, where `PID` is the identifier of the process running the task. Polling by itself is sufficient for tasks that run for more than a handful of seconds, and that do not fork themselves. To ensure that a task's resource usage is measured regardless of the polling interval and that new processes spawned by the task are also measured, we need track when task processes are created and terminate. This is done by pre-loading a library via the `LD_PRELOAD` facility from `ld.so(8)` before the Python interpreter is run. The pre-loaded library captures calls to process creation (`fork(2)`) and termination (`exit(2)` and `__attribute__(destructor)`), registering new processes to be tracked, and triggering measurements.

LFM resource monitoring is activated via a Python decorator. The decorator receives as optional arguments a dictionary that specifies the maximum resources a function may use, and a function callback that executes at the end of each polling interval. This callback can be used, for example, to report the current resources used by the function.

2) Automatically labeling resource requirements: The final missing piece is the ability to turn information about previous function resource usage into resource labels for future function execution when packing functions onto workers. If the resources consumed by function invocations were constant, this task would be easy: run the first invocation on a whole node, measure its consumption, and assign future invocations the resources used. In practice, however, resource consumption can vary significantly, and incorrect choices can reduce throughput by wasting available resources or by oversubscribing and causing job failures.

Work Queue implements an algorithm [21] that solves this problem in an automated manner with no user input. Briefly, a master runs a task under a large allocation with resource monitoring enabled. (This initial measurement can be skipped if an initial guess is manually configured, or if statistics from previous tasks are available.) The master then computes a first estimate for the resource requirements. If this initial labeling is too small and the task fails, the master updates its model and tries a larger allocation. The algorithm continues in this manner until it converges on a set of resource labels for the workload's tasks. To minimize wasted resources these labels must avoid over-fitting (resulting in many retries as tasks exceed resource limits), but must not allow allocations to become too large (resulting in unused capacity); our previous work [21] explores these trade-offs. In this work, we modified Parsl to use this algorithm for automated inference of resource labels for individual Python functions.

C. Evaluation

We tested LFM resource monitoring and labeling on the HEP, Drug Screening Pipeline, Genomic Analysis, and FaaS, workflows (see §III-B), for which workflow dependency graphs are shown in Figure 3. These workflows exhibit a variety of structural organizations and resource utilization patterns

commonly encountered in large-scale scientific applications. We carried out our evaluation at sites where these workflows are actively being used and developed, to ensure that we have a reasonable basis for comparison. For each workflow we tested four resource management strategies: perfect knowledge of the resources used (Oracle), dynamic allocation for maximum throughput (Auto), imperfect knowledge (Guess), and no-knowledge (Unmanaged). It is often not possible to obtain perfect knowledge of an application's resource usage, thus the Oracle strategy is shown only for reference. Existing state-of-the-art parallel execution frameworks (including Parsl) employ either the Guess Strategy (the user provides approximate resource information in advance) or Unmanaged strategy (tasks are allocated an entire node).

1) HEP: This workflow consists of a variable number of preprocessing, analysis, and postprocessing tasks (Figure 3: left). We ran the the HEP workflow on ND-CRC. For all tasks, the largest input is the HEP Conda environment, a 240 MB file. All tasks also access two common data files with total size of 1 MB. Data unique to each task is 0.5 MB. Each task generates 50 MB of output. Input data transfer is thus dominated by Conda environment loading, which scales with the number of workers. For 20 workers it is approximately 6 GB. Tasks run for 40 to 70 seconds.

As noted in §VI-A, it can be difficult/impossible for users to determine in advance the exact resource requirements for tasks as in the Oracle case. In the Guess configuration we run several tasks in parallel on each worker, without determining the exact requirements and attempting to maximize throughput (each task was allocated 1 core, 1.5 GB of memory, and 2 GB of disk.). The Unmanaged configuration allocates an entire worker to each task. For Oracle runs, all tasks used at most 1 core, 110 MB of memory and 1 GB of disk. The dynamic (Auto) allocation found it best to run each task with 1 core, 84 MB memory, and 880 MB disk, and then rerun the task using a full worker in case of resource exhaustion. As the workflow is uniform, less than 1% of tasks were retried because of resource exhaustion. In this experiment Auto achieves similar performance to Oracle, even when starting from no knowledge about the tasks.

We tested this scheme with a different number of tasks, number of workers, and worker sizes, with results in Figure 6. Worker nodes had 2, 4, or 8 cores, with 1 GB memory and 2 GB disk per core. Since HEP performs a significant amount of IO on its input data files, increasing the degree of parallelism on individual workers is of limited benefit. As expected, shortest completion times are achieved when perfect knowledge (Oracle) of the resources used by a task is available, with more tasks running per worker. When no knowledge is available, the automatic allocation (Auto) achieves similar completion times with less than 1% of task retries. We also show the results for imperfect knowledge (Guess), and Unmanaged resources using a full worker per task.

2) Drug Screening Pipeline: We ran the drug screening workflow on Theta, launching one worker per node. We again tested four resource strategies, with Guess runs set to 16 cores, 40 GB RAM, and 5 GB disk. We first varied the number of tasks in the workflow and ran on 14 nodes, as shown in Figure 7, left. We then fixed the number of tasks per worker to 4 and increased the number of workers (and hence the number of tasks), as shown in Figure 7, right. As expected, Oracle results in the shortest completion time, with Auto close behind. Unsurprisingly, Unmanaged has much worse performance.

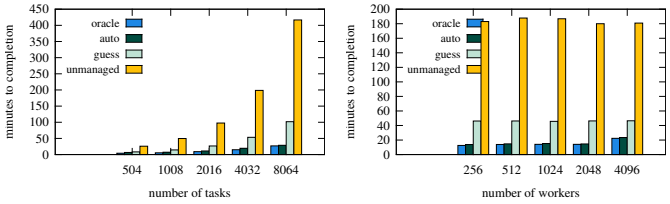

Fig. 7. Drug screening. Left: varying number of total tasks. Right: varying number of workers (workload proportional to number of workers).

3) Genomic Analysis: We ran the GDC pipeline on NSCC Aspire (Singapore), launching one worker on each 2×12-core CPUs + 96GB RAM computer node. We set resource constraints for Guess runs to 12 cores, 40 GB RAM, and 5 GB disk. First, we ran the pipeline on 14 compute nodes, and varied the number of genomes analyzed in the pipeline. Figure 8, left shows the completion time of the pipeline versus the number of genomes. We then fixed the number of genomes per worker to 1 and increased the number of workers from 1 to 16 (and hence the number of genomes), as shown in Figure 8, right. We see that Oracle leads to the shortest completion time, while Auto achieves similar completion time to Oracle. Guess and Unmanaged perform less well. Auto outperforms Oracle in a few cases, primarily because of an artifact in our Oracle setting: We manually configure the "perfect" knowledge for each type of task. In practice, the resources used by some tasks are highly dependent on the number of variants of specific genomes, which makes such perfect configurations difficult to achieve even for this work: another reason for our LFM resource management approach.

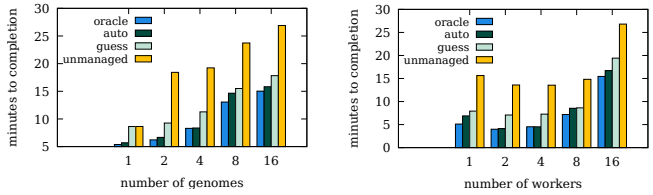

Fig. 8. Genomic Analysis. Left: varying number of total tasks. Right: varying number of workers (workload proportional to number of workers).

4) funcX: To further explore the benefits of our approach we extended the funcX FaaS service by replacing its execution components with the LFM model. funcX [22] is a distributed FaaS system that supports function execution on heterogeneous resources, including HPC clusters that use specialized HPC containers. When functions are to be executed funcX simply passes the serialized function (and its list of dependencies) to our system, using LFMs in place of containers. We note that we do not integrate static analysis or environment

distribution as these capabilities are provided by funcX. We select a FaaS benchmark [23] which classifies images using Keras ResNet [24] model. We show in Figure 9 performance for this benchmark while scaling the number of tasks (left) and workers (right) with LFMs (Auto, Guess) and without LFMs (Unmanaged). Our results show that auto labelling and LFMs results in near-oracle performance and significantly outperforms the unmanaged (non-LFM) case.

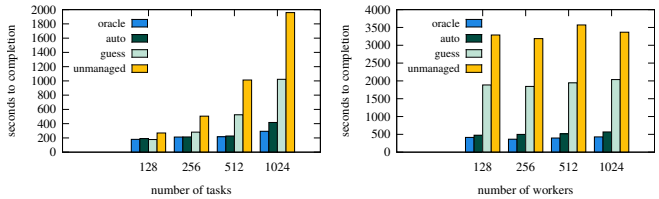

Fig. 9. funcX image classification benchmark. Left: varying number of total tasks. Right: varying number of workers (workload proportional to number of workers).

VII. Related Work

Parallel computing in Python. The Dask [1], Ray [3], FireWorks [25], and Parsl [2] libraries support distributed and parallel execution in Python. They share similar features, such as how they express parallelism in Python and their implementation of a task dependency graph to manage execution. While we focus here on Parsl, our approaches are generally applicable to all of these libraries. PyWren [26], a Python library for running Python functions on Amazon Lambda, includes a scheduler for allocating tasks to lambda functions, serialization capabilities for functions and arguments, and support for determining dependencies. The authors suggest that using Conda, as we do here, would address some of the difficulties PyWren faces in packaging some environments. Our approach further innovates with support for arbitrary HPC environments and granular resource management in LFMs.

FaaS. Commercial FaaS systems use specifications of programming functions and their dependencies to build containers for datacenter deployment. Open source FaaS systems [27] generally rely on container orchestration systems (e.g., Kubernetes) for deployment.

Containers. FaaS systems have developed new lightweight container models to address the cold start problem. Amazon's Firecracker [28] builds on KVM to implement a virtual machine monitor that provides secure isolation, low cold start latency, and low memory overhead for serverless workloads by only implementing the features that are needed for serverless container and function workloads. Similarly, SOCK [29] reduces container cold start latency by optimizing expensive operations in container initialization according to serverless needs. It uses a Zygnote process to dynamically cache popular Python libraries to reduce Python startup latency in containers. These technologies, while practical in production clusters, are unnecessarily heavyweight (e.g. security isolation) and cannot be applied to HPC sites that do not provide privileged access.

Python package management. Python virtual environments [30], enable creation of specialized software environments tailored to specific applications. Historically pip has been the de facto standard package manager for Python; however, Conda has grown in popularity and is now widely used for managing both Python and non-Python software.

Python package management on HPC. Sites often use a module system such as LMod [31] to allow users to activate a static set of software components. Many sites offer customized Conda environments optimized for HPC file systems. We leverage such Conda environments where possible. The Pynamic [32] benchmark can generate Python modules and utility libraries to test Python performance on large systems.

Python performance at scale. The Pynamic [32] benchmark can generate Python modules and utility libraries to test Python performance on large systems. MacLean et al. [6] showed that starting the Python interpreter and importing modules on many nodes places significant stress on a shared file system metadata server, and furthermore can result in poor performance for all users. They proposed to mount an image of the metadata as a local disk device to reduce the metadata server load. However, this method is not user-oriented—it requires admin privilege and is a site-specific optimization.

Adapting resources for high performance. Much research [33–37] has focused on scheduling batch jobs on HPC systems or on interference between workloads running concurrently [38–40]. Rather than looking at the job-level resource allocation problem, we focus on function-level resource matching, which has finer granularity and leads to less wasted resources. Prior resource allocation solutions generally assume that the user is familiar with job resource needs [41–46], or that the tasks submitted in a job have identical resource requirements [47]. Parallel Python systems, such as Parsl and Dask, statically divide a node's resources equally. We propose a new approach to automatically map the right resources to each function.

VIII. Conclusion

Making Python function invocations the fundamental unit of resource management in a distributed system raises issues relating to granular parallelism, management of software environments, and adaptation to computing resources. We extended and integrated Parsl and Work Queue by developing tools to handle these execution challenges automatically on behalf of the user. Our evaluation shows how our advanced dependency management and dynamic provisioning techniques can achieve near-optimal performance across a variety of computing environments and applications. Our enhancements are available to users in Parsl and Work Queue and can be installed via Pip or Conda. Test applications in Jupyter notebooks are distributed on GitHub to show how users can run complete scientific applications efficiently at scale through a purely Python interface.

ACKNOWLEDGEMENTS

This work was supported in part by NSF grants OAC-1931348, OAC-1550588, and OAC-2004894, and by the U.S. Department of Energy under Contract DE-AC02-06CH11357. The computational work for this article was partially performed on resources of the Argonne Leadership Computing Facility and the National Supercomputing Centre, Singapore.

This material is based upon work supported by the U.S. Department of Energy, Office of Science, Office of Workforce Development for Teachers and Scientists, Office of Science Graduate Student Research (SCGSR) program. The SCGSR program is administered by the Oak Ridge Institute for Science and Education (ORISE) for the DOE. ORISE is managed by ORAU under contract number DE-SC0014664. All opinions expressed in this paper are the author's and do not necessarily reflect the policies and views of DOE, ORAU, or ORISE.

REFERENCES

[1] Dask. http://docs.dask.org/en/latest/. Accessed October 2020.

[2] Y. Babuji *et al.*, "Parsl: Pervasive parallel programming in Python," in *28th International Symposium on High-Performance Parallel and Distributed Computing*. ACM, 2019, pp. 25–36.

[3] P. Moritz *et al.*, "Ray: A distributed framework for emerging AI applications," in *13th USENIX Conf. on Operating Systems Design and Implementation*, 2018, pp. 561–577.

[4] Amazon Lambda. https://aws.amazon.com/lambda. Accessed October 2020.

[5] Google Cloud Functions. https://cloud.google.com/functions/. Accessed October 2020.

[6] C. A. MacLean *et al.*, "Improving the start-up time of Python applications on large scale HPC systems," in *HPC Systems Professionals Workshop*, 2017, pp. 1–8.

[7] P. Bui *et al.*, "Work Queue + Python: A framework for scalable scientific ensemble applications," in *Workshop on Python for High Performance and Scientific Computing at SC'11*, 2011.

[8] "Coffea - Columnar Object Framework For Effective Analysis," https://github.com/CoffeaTeam/coffea.

[9] Y. Babuji *et al.*, "Targeting sars-cov-2 with ai-and hpc-enabled lead generation: A first data release," *arXiv preprint arXiv:2006.02431*, 2020.

[10] D. Weininger, "SMILES, a chemical language and information system. 1. introduction to methodology and encoding rules," *Journal of Chemical Information and Computer Sciences*, vol. 28, no. 1, pp. 31–36, 1988.

[11] R. Grossman *et al.*, "Toward a shared vision for cancer genomic data," *New England J. Medicine*, vol. 375, no. 12, pp. 1109–1112, 2016.

[12] W. McLaren *et al.*, "The Ensembl variant effect predictor," *Genome Biology*, vol. 17, no. 1, p. 122, 2016.

[13] C. Lee *et al.*, "Are user runtime estimates inherently inaccurate?" in *Job Scheduling Strategies for Parallel Processing*, 2005, pp. 253–263.

[14] T. Shaffer *et al.*, "Taming Metadata Storms in Parallel Filesystems with MetaFS," in *Workshop on Parallel Data Storage and Data Intensive Scalable Computing Systems*, 2017, pp. 25–30.

[15] W. Frings *et al.*, "Massively parallel loading," in *27th International ACM Conference on Supercomputing*, ser. ICS '13, 2013, p. 389–398.

[16] L. O. Andersen, "Program analysis and specialization for the c programming language," University of Copenhagen, Tech. Rep., 1994.

[17] P. Anderson *et al.*, "Design and implementation of a fine-grained software inspection tool," *IEEE Transactions on Software Engineering*, vol. 29, no. 8, pp. 721–733, 2003.

[18] N. Wilde *et al.*, "A reusable toolset for software dependency analysis," *Journal of Systems and Software*, vol. 14, no. 2, pp. 97–102, 1991.

[19] Conda-pack. https://conda.github.io/conda-pack/. Accessed October 2020.

[20] G. Juve *et al.*, "Practical resource monitoring for robust high throughput computing," in *IEEE Cluster Computing*, 2015, pp. 650–657.

[21] B. Tovar *et al.*, "A Job Sizing Strategy for High-Throughput Scientific Workflows," *IEEE Transactions on Parallel and Distributed Systems*, vol. 29, no. 2, pp. 240–253, 2018.

[22] R. Chard *et al.*, "funcX: A federated function serving fabric for science," in *29th International Symposium on High-Performance Parallel and Distributed Computing*. ACM, 2020.

[23] J. Kim *et al.*, "FunctionBench: A suite of workloads for serverless cloud function service," in *IEEE 12th International Conference on Cloud Computing*, 2019, pp. 502–504.

[24] K. He *et al.*, "Deep residual learning for image recognition," in *IEEE Conf. on Computer Vision and Pattern Recognition*, 2016, pp. 770–778.

[25] A. Jain *et al.*, "FireWorks: A dynamic workflow system designed for high-throughput applications," *Concurrency and Computation: Practice and Experience*, vol. 27, no. 17, pp. 5037–5059, 2015.

[26] E. Jonas *et al.*, "Occupy the cloud: Distributed computing for the 99%," in *ACM Symposium on Cloud Computing*, 2017, p. 445–451.

[27] Apache OpenWhisk. http://openwhisk.apache.org/. Accessed Oct. 2020.

[28] A. Agache *et al.*, "Firecracker: Lightweight virtualization for serverless applications," in *17th Networked Systems Design and Implementation*, 2020, pp. 419–434.

[29] E. Oakes *et al.*, "SOCK: Rapid task provisioning with serverless-optimized containers," in *USENIX Annual Tech. Conf.*, 2018, pp. 57–70.

[30] Virtual environment. https://docs.python.org/3/tutorial/venv.html. Accessed April 20, 2020.

[31] Lmod: A new environment module system. https://lmod.readthedocs.io/. Accessed October 2020.

[32] G. L. Lee *et al.*, "Pynamic: The Python dynamic benchmark," in *IEEE 10th Intl Symposium on Workload Characterization*, 2007, pp. 101–106.

[33] M. Stillwell *et al.*, "Dynamic fractional resource scheduling for HPC workloads," in *IEEE International Symposium on Parallel & Distributed Processing*, 2010, pp. 1–12.

[34] S. Herbein *et al.*, "Scalable I/O-aware job scheduling for burst buffer enabled HPC clusters," in *25th ACM International Symposium on High-Performance Parallel and Distributed Computing*, 2016, pp. 69–80.

[35] X. Yang *et al.*, "Integrating dynamic pricing of electricity into energy aware scheduling for HPC systems," in *SC'13*, 2013, pp. 1–11.

[36] M. Hovestadt *et al.*, "Scheduling in HPC resource management systems: Queuing vs. planning," in *Workshop on Job Scheduling Strategies for Parallel Processing*, 2003, pp. 1–20.

[37] Z. Zhou *et al.*, "I/O-aware batch scheduling for petascale computing systems," in *IEEE International Conference on Cluster Computing*, 2015, pp. 254–263.

[38] X. Pu *et al.*, "Understanding performance interference of I/O workload in virtualized cloud environments," in *IEEE 3rd International Conference on Cloud Computing*, 2010, pp. 51–58.

[39] Q. Zhu *et al.*, "A performance interference model for managing consolidated workloads in qos-aware clouds," in *IEEE Fifth International Conference on Cloud Computing*, 2012, pp. 170–179.

[40] X. Chen *et al.*, "CloudScope: Diagnosing and managing performance interference in multi-tenant clouds," in *IEEE 23rd International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, 2015, pp. 164–173.

[41] J. Montagnat *et al.*, "Workflow-based comparison of two distributed computing infrastructures," in *5th Workshop on Workflows in Support of Large-Scale Science*. IEEE, 2010, pp. 1–10.

[42] O. A. Ben-Yehuda *et al.*, "Expert: Pareto-efficient task replication on grids and a cloud," in *IEEE 26th International Parallel and Distributed Processing Symposium*, 2012, pp. 167–178.

[43] H. Arabnejad *et al.*, "Fairness resource sharing for dynamic workflow scheduling on heterogeneous systems," in *IEEE 10th International Symposium on Parallel and Distributed Processing with Applications*, 2012, pp. 633–639.

[44] D. Poola *et al.*, "Enhancing reliability of workflow execution using task replication and spot instances," *ACM Transactions on Autonomous and Adaptive Systems*, vol. 10, no. 4, pp. 1–21, 2016.

[45] W. Chen *et al.*, "Dynamic and fault-tolerant clustering for scientific workflows," *IEEE Transactions on Cloud Computing*, vol. 4, no. 1, pp. 49–62, 2015.

[46] I. Casas *et al.*, "A balanced scheduler with data reuse and replication for scientific workflows in cloud computing systems," *Future Generation Computer Systems*, vol. 74, pp. 168–178, 2017.

[47] N. Zakay *et al.*, "On identifying user session boundaries in parallel workload logs," in *Workshop on Job Scheduling Strategies for Parallel Processing*, 2012, pp. 216–234.