Maximizing Data Utility for HPC Python Workflow Execution

Thanh Son Phung
Douglas Thain
University of Notre Dame

Ben Clifford
Kyle Chard
University of Chicago

ABSTRACT

Large-scale HPC workflows are increasingly implemented in dynamic languages such as Python, which allow for more rapid development than traditional techniques. However, the cost of executing Python applications at scale is often dominated by the distribution of common datasets and complex software dependencies. As the application scales up, data distribution becomes a limiting factor that prevents scaling beyond a few hundred nodes. To address this problem, we present the integration of Parsl (a Python-native parallel programming library) with TaskVine (a data-intensive workflow execution engine). Instead of relying on a shared filesystem to provide data to tasks on demand, Parsl is able to express advance data needs to TaskVine, which then performs efficient data distribution at runtime. This combination provides a performance speedup of 1.48x over the typical method of on-demand paging from the shared filesystem, while also providing an average task speedup of 1.79x with 2048 tasks and 256 nodes.

ACM Reference Format:
Thanh Son Phung, Douglas Thain, Ben Clifford, and Kyle Chard. 2023. Maximizing Data Utility for HPC Python Workflow Execution. In *Proceedings of SC23 Workshop: High Performance Python for Science at Scale (SC '23).* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3624062.3624136

1 INTRODUCTION

Modern scientific workflows demand enormous compute and storage capabilities, reaching tens of thousands of CPUs and terabytes of disk space [1, 4]. For example, the Dark Energy Science Collaboration (DESC) reports an aggregated 2.5 PBs of data products resulting from processing sky images obtained from the Rubin Observatory [1]. To address this pressing need, current solutions [3, 5, 9] revolve around the "scale-out" approach: the ability to linearly scale the computational power with available resources.

To effectively utilize current solutions, users often express such applications as a dynamic workflow: necessary computations are separated and individually packaged into fine-grained tasks to be executed independently on compute nodes. In this approach, it is the responsibility of the compute node (or a proxy for that node in the case of a pilot job) to provide a task with access to its input data and software dependencies for a successful execution. It is important to note that the size of input data and software dependencies for a single task can easily reach GBs of disk space. For example, two popular deep learning frameworks `tensorflow` and `pytorch` take 2.2 GBs and 1.7 GBs of disk space, respectively. Running workflows with thousands of tasks using these frameworks implies *at least* terabytes of data transferred between the data source (e.g., a shared filesystem) and compute nodes. Such data volumes necessitate a careful decision on how to provide access to data when thousands of tasks are executed concurrently on an HPC cluster.

Several methods have been developed to provide data and dependencies to a task on a compute node. We divide these methods into two main categories:

(1) *Shared Filesystems:* Some workflow systems rely on the existence of shared filesystem mounts on compute nodes to deliver data and software dependencies to tasks. While this approach does reduce data delivery complexity for both users and developers, its performance is dependent on the performance of the shared filesystem, which can be heavily strained by other users' activities in the facility. Since production workflows are usually comprised of thousands of tasks, this problem is further exacerbated at workflow startup when all tasks ask the shared filesystem for the same set of software dependencies, overloading the metadata server and the data servers that hold the software dependencies, thus reducing the overall workflow performance.
(2) *Data Staging*: Recent efforts instead use the local storage available on compute nodes to stage data. This approach requires more work from users but can increase performance as data and software dependencies are extracted once from the shared filesystem and loaded into local storage distributed over compute nodes.

Data-staging methods can be further broken down into two technical styles:

(1) *Private Temporary Filesystems*: These solutions [11, 12] transform the aggregated local storage on compute nodes into a temporary filesystem and load data and software dependencies therein, virtually making them "closer" to compute nodes. This approach relieves the shared filesystem from its data delivery responsibility but requires that all data be transferred into the temporary filesystem upon workflow startup, and thus inducing a significant latency to a workflow execution. Furthermore, any potential awareness of data locality is hidden from all components in a workflow run due to the abstraction of a shared filesystem (a temporary filesystem is still a shared filesystem), thus removing possible data movement optimizations and sharing.
(2) *Explicit Data Placement and Management*: Workflow systems like WorkQueue[5] or Makeflow[2] instead require explicit annotations of data-to-task bindings and map locations of all data and tasks on compute nodes on-the-fly. Despite the increased burden of correct annotations on users, this dynamicity allows for several optimizations: caching common data and software dependencies at compute nodes, P2P transfer of data between compute nodes, etc.

We argue that explicit data placement and management are necessary to optimize performance of workflow execution. This is because the data-to-task annotations allow for the **maximization of data utility** in a workflow: once a data asset is loaded into a compute node, it becomes a source of data from which other compute nodes can pull from and other tasks can reuse, thus reducing pressure on the common data storage. This approach effectively views the problem of data and software dependencies delivery as a data dissemination problem: compute nodes that don't have needed data (not "infected") can pull data replicas from "infected" compute nodes instead of the shared filesystem, and tasks on the same compute nodes can reuse the same software dependencies instead of each pulling them separately.

To evaluate this idea, we integrate Parsl [3], a Python-native workflow manager that uses Python functions to define tasks and tracks task dependencies using Python futures, with TaskVine[10], a data-intensive workflow execution engine that extensively uses data-to-task annotations and local storage of compute nodes to optimize workflow executions on-the-fly. We first give a brief overview of Parsl and TaskVine, and show several technical problems in the mapping of Parsl abstractions to the TaskVine API. We then show how to overcome these problems and provide a complete data-to-task annotation to TaskVine for every task, thus enabling TaskVine to optimize workflow execution. Our work is implemented as the `TaskVineExecutor` in Parsl and can be used by production workflows by simply specifying its use in the Parsl configuration. Our results show that the execution time of a workflow with 2048 tasks can be sped up from 1.05x to 1.48x and the average task execution time can be sped up from 1.02x to 1.79x using up to 256 workers.

2 BACKGROUND

Parsl: Parsl is a Python-native parallel programming library that aims to make it easy to encode parallelism in application code. In the front end, users need only to apply Python decorators to functions that may be executed concurrently. The Parsl-provided decorators allows for optional specification of input/output file wrappers that are then staged to/from compute nodes. Once these functions are called by the Python interpreter, a Python `concurrent.futures` Future object is returned immediately, and users can check for results via the Future API (`f.done()` and `f.result()`). Parsl's core consists of two components: the *DataFlowKernel* and the *Executor*. The *DataFlowKernel*'s job is to maintain a mapping of futures to results of function calls, and the dependency call graph among called functions. Once functions' dependencies are resolved, ready functions are wrapped and given to the *Executor* as a stream of tasks to be scheduled to execute on compute nodes. The *Executor* then ensures successful executions of tasks and returns the results to the *DataFlowKernel*, which then sets the results or exceptions of Future objects appropriately.

TaskVine: TaskVine is a data-intensive workflow execution engine that follows the manager-worker paradigm. At the top layer, applications first define tasks and data on-the-fly, including software dependencies, then provide data-to-task bindings to the *TaskVine Manager*. It's important to note that data is treated as a first-class citizen just like tasks: common shareable data is defined as a unique *File* object, and the binding will then serve as a hook between registered *Files* and tasks. Once the mapping and registration are done, the *Manager* will start sending out tasks to ready compute nodes and direct the data delivery to all connected compute nodes, marking data as cacheable and transferable as appropriate. Since tasks cannot be executed before all input data and software dependencies are presented, waiting compute nodes will seek data transfers from replicas on ready nodes. Allowing waiting nodes to uncoordinatedly ask ready nodes for pending data will most likely result in bandwidth overloading between nodes in an HPC cluster and thus degrade the performance of the workflow execution and perhaps other co-located workflows. To avoid this, the *TaskVine Manager* coordinates the P2P transfers by capping the number of concurrent transfers a compute node can make (default to 3), thereby introducing an implicit throttling mechanism to the data dissemination process. A task will then start its execution when all data dependencies are staged in (all I/Os hereafter are contained in the compute node). Once all tasks finish and results are returned to the application, the *Manager* will direct workers to clean their workspaces as appropriate and free the cluster's resources.

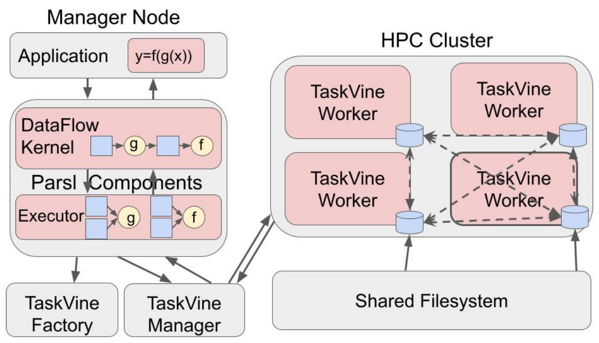


Figure 1: Parsl-TaskVine Integration Architecture

Related Work: Many workflow systems [6, 7] assume the existence of a static DAG of tasks to execute a workflow. While this assumption fits well with some applications and allows for optimizations by statically analyzing the DAG, modern applications often do not have prior knowledge of the amount of computation needed to be performed, and instead probing work is required before arriving at the formation of a DAG of tasks. Other workflow systems, such as Parsl [3] and Dask [9], allow a more dynamic expression of an application, relying heavily on Python's Future-like objects to represent tasks to be executed at compute nodes. This approach focuses on the parallelization of Python computations, but falls short in data delivery and management. They either depend on shared filesystems to deliver data and dependencies, or stream data and dependencies sequentially to each task, and thus fail to make use of readily available data replicas on other compute nodes.

3 INTEGRATION MODEL

Overview: Figure 1 shows the architecture of the Parsl-TaskVine integration. At the top level, an application defines functions to be executed concurrently and registers them with Parsl using the `app` decorator. Once these functions are called, two events happen: (1) a Future object is returned immediately per function call to the application representing a pending function execution, and (2) the function objects along with their arguments are given to Parsl's *DataFlowKernel*. The *DataFlowKernel* will add given function calls to its internal dynamic graph of computation, resolve function dependencies to determine which functions can be readily run, and send ready functions to the *Executor*. The *Executor* upon startup will spawn two processes, the *TaskVine Manager* which handles task scheduling, execution, and data management, and the *TaskVine Factory*, which exclusively manages the sizing of the pool of workers. Upon receiving the ready functions from the *DataFlowKernel*, the *Executor* then serializes the function objects and arguments using available serializing methods and sends the serialized objects to the *TaskVine Manager* process. The *Manager* process detects tasks and their associated data, registers both to its internal data structures, and annotates data-to-task bindings. The *Manager* then schedules tasks for execution in TaskVine workers (priority is given to workers that already have data dependencies available), which are launched as individual processes on compute nodes in a cluster and report for work to the *Manager* upon startup. To launch workers on an HPC cluster, we utilize the *TaskVine Factory* process, which generates helper scripts to launch TaskVine worker processes on compute nodes and automatically scales the number of workers as users wish. The *Factory* process is started by default when a workflow starts up, and will release all resources upon receiving a termination signal from the *Executor*.

Serialization/Deserialization: Our first technical problem revolves around sending Python objects (function and arbitrary argument objects) between processes (specifically, between the *Executor* and *Manager*, and between the *Manager* and *Worker*). A naive implementation can easily lead to double serialization/deserialization of every Python object per task (once per component pair), which would significantly reduce workflow performance. To avoid this, all Python objects are serialized to *TaskVine Files* in the *Executor* process, and conveyed to the *Manager* process through local file paths. The *Manager* then binds appropriate *Files* to tasks and sends a "wrapper" task instead. This "wrapper" task once sent to a worker will deserialize objects from the *File*, load them to the current Python namespace, and execute the function call normally. Results are also communicated the same way: they are serialized to a file at a worker, transferred back to the *Manager*, and deserialized and given back to the application. The Parsl-TaskVine integration uses a combination of `pickle` and `dill` [8] packages to serialize/deserialize Python objects, and can support customized serializers should users need.

Environment Packaging: To enable the comprehensive and explicit data placement and management strategy of TaskVine, users can choose to specify the necessary software dependencies. This can either be in the form of a tarball containing all software dependencies, a `conda` environment that encapsulates all software dependencies of the workflow, or a customized tarball that follows TaskVine's convention of package dependencies. If given, the *Manager* will create if needed and bind this dependency tarball to all tasks, and send the tarball over to a worker either directly or through P2P transfers. The worker caches the extracted dependencies directory using its local storage to facilitate sharing between co-located tasks, effectively localizing all file accesses to package dependencies by one large tarball transfer.

Data Caching/Sharing: Parsl-TaskVine Executor takes a proactive view on the problem of caching and sharing data. By default, all files that bind with tasks using the data-to-task bindings are cached and shared, with the expectation that such data will be used repeatedly later. By the same argument, software dependencies are

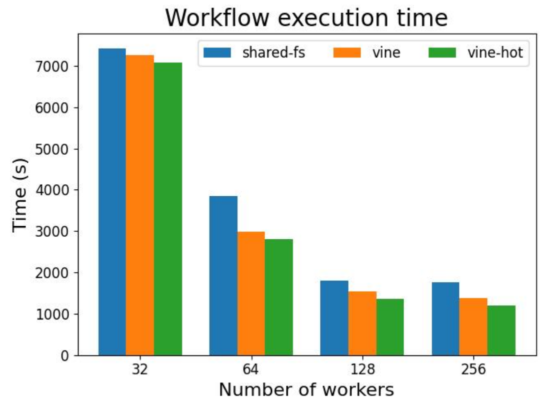


Figure 2: Workflow Execution Time

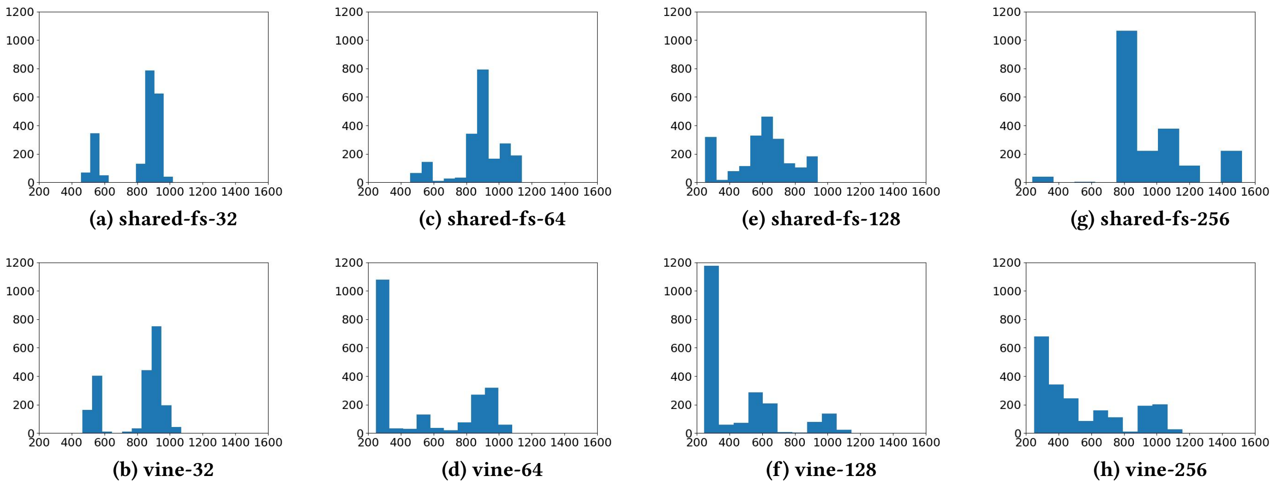
also cached and shared, as functions usually run in the same environment and thus should share dependencies. Two types of data are never cached or shared between workers: (1) computational results, as they must be returned to the application for further processing, and (2) temporary stdin, stdout, and stderr of individual tasks.

Resource Allocation and Scaling: Users have the ability to specify the amount of resources a function may consume by passing a dictionary of resource values as an argument to the function call. The Executor will detect and parse the argument if available, and use TaskVine APIs to set these resource limits accordingly. The size and number of workers can be configured and given to the *TaskVine Factory* process. Upon startup, the *Factory* process will read the configuration, create helper scripts to various batch systems, and launch and monitor workers properly.

4 EVALUATION

To evaluate explicit data placement and management in Python workflow execution, we run a hyperparameter sweep application on two configurations: (1) entirely using the `panfs` [13] filesystem, and (2) entirely using TaskVine's data distribution method. The application consists of searching a set of 2048 possible combinations of configurations of a neural network designed to categorize MNIST images. The neural network trains in approximately 5 minutes and uses at most 2 cores and 2GBs of memory and disk. Each run of a neural network training requires access to a software dependency directory of size 4.4 GBs (compressible to 908 MBs) and a dataset of 25 MBs (compressible to 17MBs). Each worker has 16 cores and 16GBs of memory and disk and thus can run 8 tasks concurrently. Finally, all workflows run on an HTCondor campus HPC cluster with varying number of workers from 32, 64, 128, to 256.

Figure 2 shows the execution time of the workflow with varying number of workers and execution modes: using the shared filesystem ("shared-fs") and using the TaskVine distribution method ("vine"). Since the software dependency and dataset tarballs can be reused over multiple runs, "vine-hot" shows the execution time of the workflow when these tarballs are cached on the manager node (these tarballs are still extracted at the compute nodes however.) We can see that the TaskVine distribution method doesn't bring major benefits when running at a small scale of 32 workers with a speedup of 1.05x. However, as we increase the number of workers to 64, 128, and 256, the TaskVine distribution method scales better


(a) shared-fs-32 (c) shared-fs-64 (e) shared-fs-128 (g) shared-fs-256

(b) vine-32 (d) vine-64 (f) vine-128 (h) vine-256

Figure 3: Histograms of task execution time in 8 workflow runs
x-axis: execution time (s), y-axis: task count.

	shared-fs			vine		
	mean	median	std	mean	median	std
32	816.7	884.6	153.7	**803.5**	885.3	171.5
64	888.6	901.8	148.6	**503.6**	296.9	299.4
128	602.8	618.2	179.2	**460.7**	317.0	229.4
256	961.3	849.2	229.9	**538.5**	437.2	265.0

Table 1: Statistics of tasks' execution time

than the shared filesystem and brings speedups of 1.38x, 1.31x, and 1.48x, respectively. We can also see the effect of diminishing returns when doubling the number of workers from 128 to 256 with the shared filesystem, only getting a 1.02x speedup. TaskVine, on the other hand, shows a better 1.15x speedup as both configurations start to be bottlenecked by the data dissemination problem.

Figure 3 shows the histograms of all 2048 tasks' execution time over 8 combinations of workflow configurations. While tasks in workflows that use the shared filesystem generally run longer due to the concurrent I/O pressure to the same data source, TaskVine's data distribution method results in a higher number of tasks finishing faster as data and software dependencies are distributed more quickly and thus reduces average task execution time.

Table 1 shows the mean, median, and standard deviation of 2048 tasks' execution time with varying combinations of the number of workers and execution modes. As we increase the number of workers from 32 to 256, TaskVine's distribution method yields better average task execution time, obtaining speedups of 1.02, 1.76, 1.31, and 1.79, respectively.

5 CONCLUSION

We intergrated Parsl with TaskVine to utilize local storage of compute nodes and maximize data utility in Python workflow executions. We showed that by using an explicit data placement and management strategy, TaskVine can coordinate and transmit data and software dependencies faster than using a shared filesystem at scale, obtaining a speedup of 1.48x and 1.79x in workflow execution time and average task execution time, respectively.

REFERENCES

[1] Bela Abolfathi, David Alonso, Robert Armstrong, Éric Aubourg, Humna Awan, Yadu N Babuji, Franz Erik Bauer, Rachel Bean, George Beckett, Rahul Biswas, et al. 2021. The lsst desc dc2 simulated sky survey. *The Astrophysical Journal Supplement Series* 253, 1 (2021), 31.

[2] Michael Albrecht, Patrick Donnelly, Peter Bui, and Douglas Thain. 2012. Makeflow: A portable abstraction for data intensive computing on clusters, clouds, and grids. In *Proceedings of the 1st ACM SIGMOD Workshop on Scalable Workflow Execution Engines and Technologies*. 1–13.

[3] Yadu Babuji, Anna Woodard, Zhuozhao Li, Daniel S. Katz, Ben Clifford, Rohan Kumar, Lukasz Lacinski, Ryan Chard, Justin M. Wozniak, Ian Foster, Michael Wilde, and Kyle Chard. 2019. Parsl: Pervasive Parallel Programming in Python. In *Proceedings of the 28th International Symposium on High-Performance Parallel and Distributed Computing* (Phoenix, AZ, USA) *(HPDC '19)*. Association for Computing Machinery, New York, NY, USA, 25–36. https://doi.org/10.1145/3307681.3325400

[4] Jakob Blomer, Philippe Canal, Axel Naumann, and Danilo Piparo. 2020. Evolution of the ROOT tree I/O. In *EPJ Web of Conferences*, Vol. 245. EDP Sciences, 02030.

[5] Peter Bui, Dinesh Rajan, Badi Abdul-Wahid, Jesus Izaguirre, and Douglas Thain. 2011. Work queue+ python: A framework for scalable scientific ensemble applications. In *Workshop on python for high performance and scientific computing at sc11*.

[6] Ewa Deelman, Karan Vahi, Gideon Juve, Mats Rynge, Scott Callaghan, Philip J Maechling, Rajiv Mayani, Weiwei Chen, Rafael Ferreira Da Silva, Miron Livny, et al. 2015. Pegasus, a workflow management system for science automation. *Future Generation Computer Systems* 46 (2015), 17–35.

[7] Paolo Di Tommaso, Maria Chatzou, Evan W Floden, Pablo Prieto Barja, Emilio Palumbo, and Cedric Notredame. 2017. Nextflow enables reproducible computational workflows. *Nature biotechnology* 35, 4 (2017), 316–319.

[8] Michael M McKerns, Leif Strand, Tim Sullivan, Alta Fang, and Michael AG Aivazis. 2012. Building a framework for predictive science. *arXiv preprint arXiv:1202.1056* (2012).

[9] Matthew Rocklin. 2015. Dask: Parallel Computation with Blocked algorithms and Task Scheduling. In *Proceedings of the 14th Python in Science Conference*, Kathryn Huff and James Bergstra (Eds.). 130 – 136.

[10] Barry Sly-Delgado, Thanh Son Phung, Colin Thomas, David Simonetti, Andrew Hennesse, Ben Tovar, and Douglas Thain. 2023. TaskVine: Managing In Cluster Data for High Throughput Data Intensive Workflows. *WORKS Workshop on Workflows in Support of Large Scale Science at Supercomputing* (2023).

[11] Osamu Tatebe, Kazuki Obata, Kohei Hiraga, and Hiroki Ohtsuji. 2022. Chfs: Parallel consistent hashing file system for node-local persistent memory. In *International Conference on High Performance Computing in Asia-Pacific Region*. 115–124.

[12] Marc-André Vef, Nafiseh Moti, Tim Süß, Tommaso Tocci, Ramon Nou, Alberto Miranda, Toni Cortes, and André Brinkmann. 2018. Gekkofs-a temporary distributed file system for hpc applications. In *2018 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 319–324.

[13] Brent Welch and Garth A Gibson. 2004. Managing Scalability in Object Storage Systems for HPC Linux Clusters.. In *MSST*. Citeseer, 433–445.