TaskVine: Managing In-Cluster Storage for High-Throughput Data Intensive Workflows

Barry Sly-Delgado
University of Notre Dame
South Bend, Indiana, USA

Thanh Son Phung
University of Notre Dame
South Bend, Indiana, USA

Colin Thomas
University of Notre Dame
South Bend, Indiana, USA

David Simonetti
University of Notre Dame
South Bend, Indiana, USA

Andrew Hennessee
University of Notre Dame
South Bend, Indiana, USA

Ben Tovar
University of Notre Dame
South Bend, Indiana, USA

Douglas Thain
University of Notre Dame
South Bend, Indiana, USA

ABSTRACT

Many scientific applications are expressed as high-throughput workflows that consist of large graphs of data assets and tasks to be executed on large parallel and distributed systems. A challenge in executing these workflows is managing data: both datasets and software must be efficiently distributed to cluster nodes; intermediate data must be conveyed between tasks; output data must be delivered to its destination. Scaling problems result when these actions are performed in an uncoordinated manner on a shared filesystem. To address this problem, we introduce TaskVine: a system for exploiting the aggregate local storage and network capacity of a large cluster. TaskVine tracks the lifetime of data in a workflow –from archival sources to final outputs– making use of local storage to distribute, and re-use data wherever possible. We describe the architecture and novel capabilities of TaskVine, and demonstrate its use with applications in genomics, high energy physics, molecular dynamics, and machine learning.

ACM Reference Format:
Barry Sly-Delgado, Thanh Son Phung, Colin Thomas, David Simonetti, Andrew Hennessee, Ben Tovar, and Douglas Thain. 2023. TaskVine: Managing In-Cluster Storage for High-Throughput Data Intensive Workflows. In *Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis (SC-W 2023), November 12–17, 2023, Denver, CO, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3624062.3624277

1 INTRODUCTION

Many scientific applications can be expressed as high-throughput workflows that combine data assets and executable tasks into large graphs that can be executed in parallel on distributed systems such as university clusters, commercial clouds, and leadership HPC machines. A wide variety of workflow management systems [2, 5, 7, 13, 26, 28] have arisen to express workflows in various domains and programming languages.

For many such workflows, the primary limitation is not the peak performance of computation at steady state, but rather the costs of managing data throughout execution. This includes the startup cost of deploying the assets needed for execution at scale, managing intermediate data generated between task execution, and retaining reusable data at the site of execution. A simple application written in a dynamic programming language may require a specific language runtime, a fleet of supporting libraries, and various configuration files that tie the bundle together. More complex applications may tie together scripts, executables, and libraries drawn from multiple language environments. These may come in the form of containers [22, 25] or shared packages in the filesystem [14, 17, 21]. Datasets to be consumed by the application must be downloaded, uncompressed, prepared, and arranged for distribution or partitioning as required. These intermediate steps present a *usability* problem because end users must struggle to identify and deploy the needed assets at every new site. But they also present a *performance and resource management problem* because the distribution of such assets may be a significant fraction of the time and resources needed to execute the application itself.

A more global approach is needed to address such data-intensive workflows. To improve usability and portability, a complete workflow should encompass the original archival data sources of data and software and the preparatory steps needed to put them into action. Doing so should not require continuous communication with the archive, but rather the system should be capable of capturing and naming external data assets in a reproducible way. To improve deployment performance, the internal storage resources of cluster nodes can be harnessed to share and distribute common assets in a scalable way, instead of depending on a single shared filesystem which may present a bottleneck at scale. Finally, the limited resources of local storage and network bandwidth must be carefully managed to ensure that parallel actions do not overwhelm the available capacity.

TaskVine is a workflow execution system that puts these concepts into practice. A TaskVine workflow consists of a dynamic graph of data items which are mounted into a private namespace for each task. The initial inputs of a workflow are best expressed

ACM ISBN 979-8-4007-0785-8/23/11…$15.00
https://doi.org/10.1145/3624062.3624277

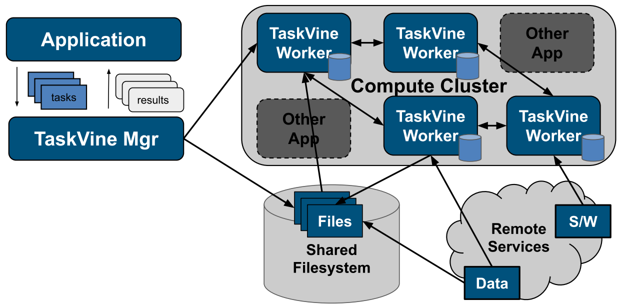


Figure 1: TaskVine Architecture
An application uses the TaskVine API to specify the relationships of tasks and data. The TaskVine Manager program coordinates multiple workers running in a cluster to manage local storage, send and receive files, execute tasks, and report completion.

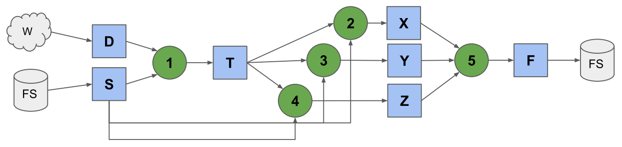


Figure 2: Example Workflow
A TaskVine workflow consumes input datasets from archival sources or a shared filesystem, produces outputs to be consumed by other tasks, and then only places final outputs back in a reliable shared filesystem. Common inputs (like software S) are efficiently replicated while intermediate files (like file T) exist only in ephemeral storage.

as archival sources of software and data, which are transformed as needed until ready for task consumption. A TaskVine workflow is executed on a set of workers that exploit the local storage, memory, and compute capabilities of cluster nodes. A manager process schedules data items and tasks to the workers, seeking to carefully manage the limited storage and network resources of the cluster. Frequently used items such as software packages and reference datasets are kept within the cluster for reuse across similar workflows, thus removing substantial load from the shared filesystem.

In this paper, we present the architecture and programming model of TaskVine, and detail the key mechanisms and strategies for managing data assets within the cluster: data transformation from archival sources, storage management on worker nodes, network management across the cluster, and the serverless computing model. We describe how TaskVine has been applied to four distinct applications: high throughput genome search (BLAST), data analysis in high energy physics (TopEFT), AI-guided molecular simulation (Colmena), and machine learning model optimization (BGD). In each case, we demonstrate how these workflows exploit the TaskVine mechanisms to make effective use of in-cluster storage resources.

2 ARCHITECTURE

2.1 Overview

Figure 1 shows the general architecture of TaskVine. TaskVine seeks to effectively manage the local storage on each node in a cluster, arranging for tasks to execute in close proximity to the data that they consume and produce. From a user's perspective, they declare the data objects and tasks that comprise a workflow, and they are notified as tasks complete and outputs are produced. Wherever possible, data is left in place where it is created, moved only as needed to satisfy replication or output constraints, and reused between common tasks as much as possible. The scheduling objective is to replicate and place data first, and then schedule tasks within the constraints of available data.

Each TaskVine worker is responsible for managing the resources on a single node: CPUs, GPUs, memory, and storage. Worker storage (whether HDD, SSD, or NVMe) is organized as a flat cache of data objects, each with a unique name assigned by the manager. The worker tracks the size and resources available in the cache, and informs the manager of every status change of interest. To prevent cached files from using up all the disk space of a worker's local filesystem, the manager tracks which data belongs to which task and can either delete or relocate that data to another worker, if appropriate. The worker also manages a queue of pending transfers assigned by the manager and can either fetch data from remote data services or from peer workers. Transfers are supervised by the manager to avoid contention. Distributing assets between nodes resembles the construction of a peer-to-peer network. File transfer protocols such as BitTorrent have been evaluated on HPC clusters with mixed results [16]. One conclusion from this study is that the unmanaged transfers lead to hot spots in the network, while "fairness" measures make little sense in a cluster of cooperative nodes. As we will show, managed peer-to-peer transfers result in a stable and performant TaskVine system.

Each task in TaskVine represents a unit of execution that must be scheduled with respect to the data that it consumes and produces. To manage these relationships, TaskVine requires users to explicitly bind each task to its inputs and outputs. When data sources are used repeatedly by similar tasks, multiple immutable replicas are made across workers, and then implicitly shared by all tasks within a given worker. Tasks come in several varieties: A plain **Task** indicates a Unix command line executed in a private sandbox directory; a **PythonTask** indicates an invocation of a function accompanied by a self-contained execution environment; a serverless **FunctionCall** indicates a remote invocation of a separately defined and executed **LibraryTask**. Each of these task modalities may be mixed within a single workflow. As we show below, an application can combine together traditional executable programs with lightweight serverless invocations in one workflow.

Reliable high-throughput execution of large workflows requires efficient resource management. Each task is defined to consume a fixed quantity of resources (cores, memory, disk) which are monitored and enforced at execution time. If a task exceeds the declared allocation, it is returned to the manager. Depending on the user's configuration, the manager will either execute it elsewhere with a larger allocation, or return it to the user as a failure. This permits the worker to reliably "pack" a large number of small concurrent tasks on to a node at once without overcommitting and risking the failure of all tasks. In a similar way, storage resources are enforced

```python
import taskvine as vine
m = vine.Manager()
blast_url = m.declare_url("https://.../blast.tar.gz")
blast = m.declare_untar(blast_url,cache=worker)

land_url = m.declare_url("https://.../landmark.tar.gz")
land = m.declare_untar(land_url,cache=workflow)

for i in range(1000):
  query = m.declare_buffer(make_query(i),cache=task)
  t = vine.Task("blast/bin/blast -db landmark -q query")
  t.add_input(query, "query")
  t.add_input(blast, "blast")
  t.add_input(land, "landmark")
  t.add_env("BLASTDB","landmark")
  m.submit(t)
```

Figure 3: Example TaskVine Application

A brief example in which 1000 tasks are generated. Each task shares a common software package (`blast`*) and dataset (*`land`*) provided by archival sources, and also has a unique buffer input (*`query`*) provided by the manager. All three are presented as files in the task sandbox.*

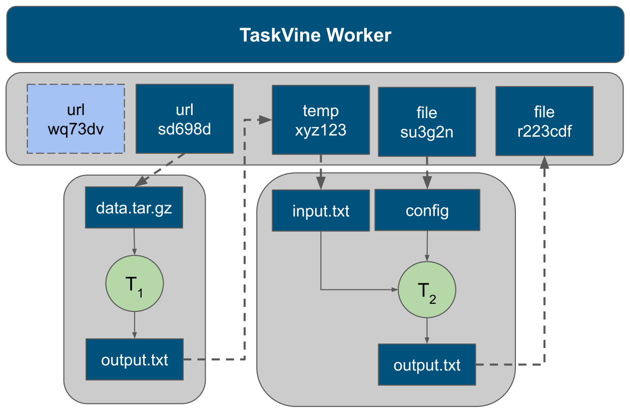


Figure 4: Worker Storage Management

Each worker maintains a cache of stored objects, each with a unique generated name. Remote assets are downloaded on demand when a task requires them. Each task is executed in a sandbox that maps unique names into a local namespace. Task T_1 reads `url-sd698d` *as* `data.tar.gz` *and produces* `output.txt` *which becomes* `temp-xyz123`*. Task T_2 later consumes that files as* `input.txt`*.*

at the worker and controlled by the manager, including the distribution and assignment of data, cache admittance and eviction of a worker's persistent cache.

2.2 Components

Figure 2 shows an example of a TaskVine workflow, which consists of an application submitting tasks to be deployed by a single manager and executed on a pool of workers.

The **application** is a program written using the TaskVine API which declares the files and tasks that form the workflow. Each source data item must be declared to the manager, indicating its original source. Each task to be executed must also be declared to the manager with the necessary input and output files to be attached to the task. Figure 3 shows an example of constructing a simple workflow consisting of software and data drawn from an archival source in order to run 1000 tasks that each perform queries against a BLAST [24] genomics database. If known, the entire workflow can be stated by the user all at once, or the task graph can be built incrementally, based on outside information or results returned from completed tasks. The TaskVine API can be used to write custom applications in C or Python or as a lower execution layer for a higher level workflow system, such as Parsl [7] or Coffea [31].

The **manager** directs the overall execution by accepting the workflow definition, dispatching tasks to workers, directing file transfers to/from workers, collecting results, and performing garbage collection. As a general rule, the manager directs all *policy* decisions, while the worker provides the *mechanisms* for execution. For example, the manager dispatches tasks to specific workers, which execute them asynchronously; the manager directs files to specific workers, but the workers transfer them asynchronously. The manager collects reports from each worker about its available resources, running tasks, cached data, and status of file transfers. As a result, the manager has a detailed picture of the distributed state of the system to make informed decisions, such as placing tasks based on data locality, scheduling transfers to avoid contention, and duplicating items for reliability.

The **worker** receives instructions from the manager to execute tasks in isolation, manage local storage, and perform file transfers asynchronously. Figure 4 shows the basic structure of a worker. A cache directory contains all of the data objects stored on local disk, each with a unique *cache name* is generated by the manager. Files to be transferred from remote sources are held in a pending state and downloaded by the worker to satisfy task needs. Each task is executed in a sandbox with a private namespace, with each input and output file linked in using a user-readable name. The sandbox is deleted when the task completes, so the only persistent data objects are those explicitly extracted from the completed task. Commonly used and shared data items may persist in the cache to be used by future task executions, and are only deleted at the manager's direction. Workers may join and leave the system dynamically as cluster resources change availability.

2.3 Data Definitions

All data accessed or produced by a TaskVine workflow must be explicitly declared by the application, so that it can be properly transferred and presented to each task. For brevity, each named data object in TaskVine is known as a **File**, whether it is a single file, a large container image, or a directory hierarchy with millions of entries. A File may be one of several subtypes: a **LocalFile** which names a file or directory in the shared filesystem of the cluster; a **BufferFile** which is a (typically) small unit of literal data in the application's memory space; a **URLFile** which is a reference to a remote data object; a **TempFile** which names an ephemeral file that exists only temporarily within the cluster and is never materialized outside. In the case of remote files such as URLFiles and TempFiles,

```python
l = m.create_library("/path/to/opt.py", "optimizer")
m.install_library(l)

for i in range(1000):
    t = vine.FunctionCall("optimizer", "gradient", i)
    m.submit(t)
```

Figure 5: Serverless Task Declaration
This application creates a Library object by giving the path to a Python module and naming it `optmizer`*. A FunctionCall task names the Library, a function within the library, and the arguments. The task is then dispatch to invoke the deployed Library on a worker.*

```python
def declare_xrootd( url, proxy ):
  t = vine.Task("xrdcp {} output".format(url))
  t.add_input(proxy,"proxy509.pem")
  t.set_env("X509_USER_PROXY","proxy509.pem")
  t.add_output(m.declare_temp(),"output"))
  return m.declare_mini_task(t)

p = declare_file("proxy.file",cache=task)
f = declare_xrootd("xrootd://server/path",p)
```

Figure 6: MiniTask Definition
A MiniTask defines a program to be executed on demand to generate a file at the worker. This example shows a MiniTask defined to add support for XRootD data transfers with user provided credentials.

simply declaring the file does not mean it exists yet at the worker. In these cases, the worker must fetch the URL or create the temporary file sometime after the task is sent by the manager. Therefore, when the worker does acquire the file, it will send an asynchronous **cache-update** message to inform the manager that the file is present for scheduling purposes.

TaskVine files are *immutable*: once created (or transferred), their contents do not change, thus allowing file replication to needed tasks without further consistency checks. A given file may exist on multiple workers simultaneously as needed to satisfy tasks. As described below, the manager generates a unique name for each file to ensure that common files are discoverable and reusable. The application may offer the manager cache hints about the expected lifetime of each file. A cache lifetime of **task** indicates that the file will not be consumed after the task completes, and it can be discarded immediately. A cache lifetime of **workflow** (the default) indicates that the file may be re-used multiple times during the current workflow run, but may be deleted at its conclusion. A cache lifetime of **worker** indicates that the file will be used again by future workflows, and may be kept on the worker as long as resources are available. This is typically used for common software packages and reference datasets, and requires additional effort by the manager to generate persistent names. In Figure 3, note that the `blast` software has a cache hint of `worker` because it can be used by many different workflows, while the per-task query text is `task`, because it is needed by that task only.

2.4 Task Definitions

A plain **Task** describes a Unix program and command line arguments to be executed in a private namespace by a worker. All data needed by the task must be explicitly described: each input file that it requires must be added with `add_input` and connected to a data source. Every output file that it produces must be described with `add_output` and connected to a data sink. The executable program and any libraries or other dependencies needed must be delivered explicitly via input files. (For example, see the invocation of `blast` in Figure 3.) If known, the resources needed by the task (cores, memory, gpus) should also be described so that the manager can make appropriate placements. A variety of optional details may be given to each task to modify fault-tolerance, error propagation, resource management and monitoring. Several specialized task types are then derived from this basic abstraction.

A **PythonTask** is a specialization of a plain Task to execute a self-contained Python code. Rather than invoking a Unix command line, a PythonTask names a function (in the body of the application) and arguments to that function. The function code is serialized along with the needed Python dependencies, which are sent as inputs for the task. From there, the PythonTask invokes the python interpreter, loads the necessary data and function code, and executes it. This relieves the user of some of the complexities of managing the library environment, keeps them within the Python programming space, and still makes available all of the other features surrounding task management.

For many data analysis applications, the overhead of packing, sending, and setting up execution environments at a worker node may be a significant fraction of the total runtime. TaskVine addresses this overhead by providing a serverless computing model that allows the reuse of execution environments over many short running tasks. This model consists of two parts: a LibraryTask and a FunctionCall. A **LibraryTask** contains arbitrary user-defined functions, and is "installed" once by the application, and then transparently deployed to workers where it runs continuously. To invoke the functions in the Library, a **FunctionCall** is used which replaces the UNIX command of a regular task with the name of a Library function to run. Figure 5 shows an example manager program using the serverless model by creating LibraryTask from a executable and then creating FunctionCall Tasks to run a function called function_name included in that Library.

In many cases data and software needs to be prepared or configured in some way *on the worker* prior to direct use by a task. This might be as simple as uncompressing a stream with `gzip` or as complex as recompiling a package to take advantage of local accelerator architectures. We observe that such transformations require all the abstractions of a task: a command, input files, output files, and time and resources needed to perform the transformation. And so, we define a **MiniTask** to be a task definition that is executed on demand in order to produce a File object on a worker as needed. TaskVine provides wrappers for built-in MiniTasks that perform common operations such as packaging and compression. For example, in Figure 3, the `declare_untar` call indicates a MiniTask which unpacks the given url and returns the uncompressed output.

MiniTasks provide a natural way of expanding the capabilities of TaskVine in a precise manner. Figure 6 shows how to define

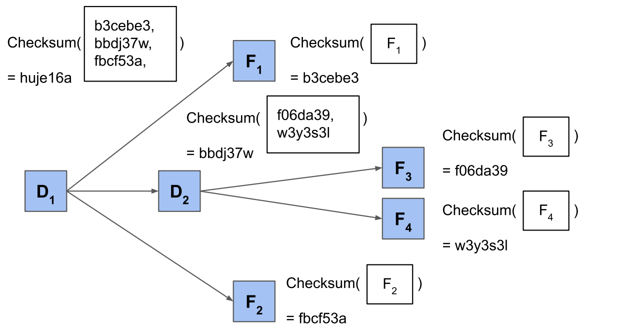


Figure 7: Directory Cache Name Merkle Tree

Cache names for files are generated by hashing the contents of a file. For a directory, we recursively hash the contents of the directory. A directory's cache name is generated by its content's cache names

a MiniTask that adds the capability to transfer input data via the XRootD [15] system used in high energy physics. A transfer requires the `xrdcp` executable to be invoked, along with an X509 proxy credential provided by the user, and an environment variable set appropriately. The user naturally desires that the credential should not be cached indefinitely, but the data so obtained may be. Once defined this way, the transfer method becomes a precise component of the workflow: data produced is assigned a unique name, cached at workers, and shared among tasks like any other file.

3 TASKVINE IMPLEMENTATION

3.1 Environment Management

TaskVine provides the ability for users to provide and manage varying execution environments at the task level. Task environments may include specific libraries, containers, databases and more. Environments often require some startup cost usually in the form of staging files, or linking libraries. In the scenario where tasks are not sharing environments, this cost of setup becomes a sizeable amount of the total runtime of a workflow. From TaskVine's perspective, the necessary steps to setup a task's environment can be seen as a task itself. This task has a explicit input requirement such as a tarball or a container. It also requires the command detailing how the inputs should be prepared. The resulting output is a ready environment that can be used by multiple tasks. MiniTasks address this need for an abstraction that properly stages data for tasks. Aside from TaskVine's built-in MiniTasks, user's can declare their own MiniTasks which sits in a space in between a task and a file. MiniTasks require users to explicitly detail its command, input files, and output files. The value returned by a MiniTask is a standard file object that can be used as an input.

3.2 Storage Management and Naming

Effective storage management is key to the performance of TaskVine. In many workflows the same data items will be reused many times, both within the same workflow and across multiple workflow executions. Reuse is both spatial and temporal. Spatially, multiple tasks running concurrently on the same worker should share the same immutable input files, thus avoiding unnecessary transfers and duplicate storage. Temporally, subsequent tasks running on the same worker should be able to share the prior input files, and where possible, consume the outputs of prior tasks. Thus, the worker must have a persistent storage cache that can serve multiple workflows and minimize the movement of data to/from the worker.

To implement a persistent cache that will serve multiple workflows, files within the cache must be named consistently to ensure the accurate execution of an application. Simple user-visible file names are not enough: it is all too easy for two applications to give the name `data.txt` to different content. Instead, the manager is responsible for giving a unique **cache name** to each file. The scope of this name depends upon the maximum lifetime of the file stated by the application. Files with cache lifetime of **task** or **workflow** are only visible within the context of a single workflow and will never be reused across workflows. In this case, the manager internally generates a random name, and ensures that no two names within a single run of a workflow collide. These files are automatically deleted at the conclusion of a workflow, thus avoiding the possibility of polluting a future run that might choose the same random names. However, files with the cache lifetime of **worker** need a perpetually unique cache name, because they are retained by the worker when a workflow completes, and may be shared across multiple workflows controlled by distinct managers. Our general approach is to use **content addressable names** that are computed from the content of a file, and therefore consistent across workflow executions. However, there is some expense to producing such names, and thus some variation across file types.

A **LocalFile** sends a local file or directory as input. A plain file is hashed using the standard MD5 checksum to create a cache name. For directories, the contents of the directory must be hashed recursively to create a cache name, as shown in Figure 7. Each file in the tree is hashed as normal using the MD5 algorithm. Then, each directory is treated as a small document consisting of the names of files (and directories) and their metadata. This document is then hashed to produce a single name for the entire directory.

A **BufferFile** consists of the content of a memory buffer in the manager to be sent as an input file. This cache name is computed by hashing the buffer contents when it is attached to a task.

A **URLFile** represents a remote URL for the worker to download asynchronously and make available to a task when needed. This presents a naming problem because the manager does not have direct access to the contents of the file, and downloading for that purpose would harm performance at workflow construction time. However, the manager is able to retrieve the HTTP header from the URL and use the file metadata to generate a strong cache name. In the ideal case, the header already contain an MD5 or SHA-1 checksum and the manager can employ this as the stable cache name. However, this is common only for libraries and other archival institutions. Alternatively, the manager will construct a cachename by combining the URL address with the **E-Tag** and **Last-Modified** elements of the header, and hash those to produce a cache name. While this does not produce a true content-derived name, these header elements are guaranteed to change if the content of the URL changes, and thus avoid any problem of stale data. In the unlikely event that none of these header fields are present, the manager will download the file content, and generate a hash from the local copy.

A **MiniTask** is a file produced on demand by a task specification. Because the content of a MiniTask is unknown prior to its execution, it cannot simply be named by its content. Instead, its cachename is computed from the Merkle tree of the task specification, encompassing the task's command, resources, and cache names of its input files computed recursively. In a similar way, A **TempFile** is an ephemeral file that is the output of a normal task, and is also named by computing the hash of the producing task.

3.3 Transfer Management

Decisions about transfer methods and data distribution have a profound effect on the outcome of a workflow, especially in cold-start situations where caching cannot yet be utilized. Since data movement is a direct consequence of scheduling tasks, transfer management is tightly coupled with the TaskVine scheduler.

In order to provide coordination, the manager must be able to locate files in the cluster and track the movement of data. Files are located at workers by the manager through the internal **File Replica Table**, which presents a unified view of the cluster storage consisting of the files present at all connected workers. The movement of data is monitored through the manager's **Current Transfer Table**. Each scheduled file transfer is stored at the manager with a UUID that the worker will respond with in its cache update message, indicating the transfer has completed. This allows the scheduler to observe how many concurrent connections a source is supporting, in turn allowing the use of defined scheduling limits to avoid hotspots.

The user initiates the management of data as soon as they declare data dependencies for a task. Each declaration creates a type of file that may be attached to a task. Before the manager dispatches a task to a worker, it must know the worker is in possession of any input file dependencies. The manager will send the file to the worker, or instruct the worker on how to obtain it. It is in this period of time that the scheduler is offered decisions over task placement and transfer management.

An effective scheduler in this context must assign data and tasks to workers while considering several issues that are in inherent tension. We aim to efficiently reuse data, however we also wish to duplicate data to increase concurrency. We provide the ability to use workers as sources of shared input files, yet we must not abuse a specific source. To address these considerations, we have developed a *conservative* scheduling strategy that respects the basic system design principles as follows:

Tasks are scheduled primarily to match the cached files present at each worker, where a worker who possesses the most dependencies for a task will be chosen to receive the task. In the case where an optimal worker is not available, we assign the task to an arbitrary worker, and move on to scheduling file transfers.

In the case where files are not present on the worker, file transfers are scheduled by the manager shortly before task dispatch. A task will have a "fixed" source for its input files, such as a remote-URL or the manager. For every input file in the task, the manager will consider where the file is currently replicated at other workers in the system using the File Replica Table. If the file is located on another worker, and this worker is not currently over the provided

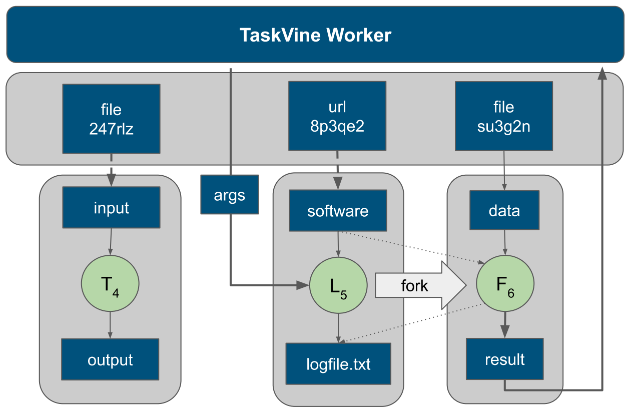


Figure 8: Serverless Execution

A normal task T_4 executes alongside a Library Instance L_5 and FunctionCall F_6. Library Instance F_5 is a Task that brings data assets as normal, but runs continuously, waiting for invocations. FunctionCall F_6 is invoked by the worker sending the function arguments to Library Instance L_5, which forks and then begins running the already-loaded code. The results of F_6 are returned as a normal Task.

transfer limit, the manager will modify the task description, directing the target worker to retrieve the file from the new source. This conservative approach always prioritizes worker transfers over the original task description. In the case where there is no opportunity to schedule a worker transfer, we consult the defined limit on concurrent transfers from the original source, being the remote-URL or the manager. The limits defined on concurrent transfers from the original source and each worker are configurable by the user.

Future considerations for scheduling optimization include decisions based on task execution time, bandwidth at specific workers, and the corresponding cost of transferring a dependency over the network. If a task is short-running, yet it depends on a complex environment that is created on the worker, it may be better to wait for a worker to become available that has the dependencies already in its cache. While scheduling files, we may be presented with a set of workers who are available to share an input file. Rather than choosing the first available worker, it may benefit us to collect some information about previous interactions to select a source that has a history of high-bandwidth transfers.

3.4 Serverless Computing

In many scientific workflows, similar tasks are executed many times with slight variations in input parameters. This can result in an excessive duplication of the same initialization work, such as starting a container, loading common libraries, or reading a dataset from storage. In extreme cases, the initialization overhead can become a significant portion of the overall workflow, as seen in Figure 9. This problem is amplified on a shared file system when many tasks access the same files and everything slows to a crawl.

Figure 8 shows the TaskVine serverless [6, 11, 18, 19] computing model, which enables low latency function invocations alongside regular tasks. Serverless tasks prevent repeatedly performing initialization work by creating persistent processes on a worker that exist over the course of a workflow. The persistent processes mean that expensive operations, like reading a data set into memory, only need to be performed once per worker, and then future invocations of that task can be performed without that overhead.

This requires coordinating two specialized task types: LibraryTasks and FunctionCall tasks. A **LibraryTask** is a task which runs a **Library**, which is an arbitrary program containing a collection of functions which can be invoked by the worker. The manager installs a Library onto a worker by sending out a LibraryTask, which may be accompanied by dependent files and other resources like any other task. After receiving the LibraryTask, the worker creates a pipe to communicate with the Library and forks to start it. The worker then waits for the Library to send a JSON initialization message describing its functions and capabilities.. Once this is complete, the process that is running the Library is called a **Library Instance**, and this Library Instance passively waits to receive messages from the worker. In order to invoke one of the functions on the Library, the worker uses the Library protocol, which involves sending a JSON invocation message describing the name of the function to execute as well as the arguments for that invocation.

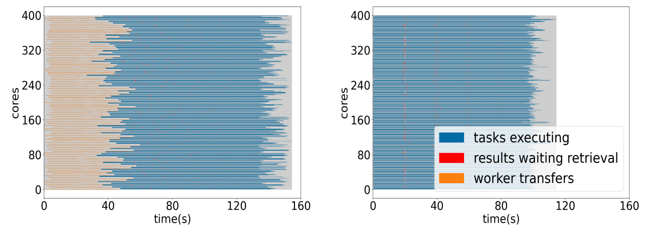
FunctionCall tasks are used to perform these invocations. FunctionCall tasks are like regular tasks but name function to run with serialized arguments, instead of a having command line with arguments. FunctionCall tasks are sent to a worker like normal tasks, but after the worker receives a FunctionCall Task it communicates with the LibraryInstance using the Library protocol to invoke the specified function. Once the Library Instance receives the invocation, the Library Instance forks to run that function with the given arguments. The forked process completes the task and returns the result to the Library Instance over a pipe. The Library Instance then passes that task output back to the worker, and this is used as the output for the FunctionCall Task. After completing the FunctionCall Task, the Library Instance returns to passively waiting for the next invocation request from the worker.

The creation and installation of Libraries is handled by the manager. The manager creates LibraryTasks given a name and a regular task, where the command line executable of the task is the Library. LibraryTasks can also be created from a list of Python functions which are packed into a Library along with all of their dependencies. LibraryTasks can be installed after creation, which begins the process of distributing LibraryTasks to workers. The manager will continue sending LibraryTasks to workers until each worker has been sent the Library. These Library Instances exist on the workers until the LibraryTask is removed or the workflow ends.

Resource management for LibraryTasks and FunctionCalls is handled largely the same as conventional tasks, allowing them to co-exist with other task types. Each Library instances consumes a static resource allocation (CPUs, GPUs, memory, storage) on each worker it runs on defined by the resource allocations for the LibraryTask. The worker has those resources set aside for running the Library Instance until it is removed. Each FunctionCall task also consumes a set of resources in addition to the LibraryTask. Tasks of all types can then be packed into a worker in the same way.

4 EVALUATION

We evaluate the effectiveness of the TaskVine mechanisms first through targeted experiments on the key features, and then by demonstrating four complete scientific applications. All evaluations are performed on a 20K-core shared university cluster managed by HTCondor. Cluster nodes are a mixture of hardware types, ranging from 12-64 cores, 16-256 GB RAM, and 50GB-2TB of local SSD storage. 10Gb Ethernet is used throughout. The shared filesystem is a Panasas [36] cluster consisting of 3 metadata nodes and 12 storage nodes providing 912TB storage with 30K metadata ops/s, and 5GB/s data throughput. Workflows are executed by submitting TaskVine workers of the desired size as batch jobs, and then starting the manager and application on the cluster head node.



a. Cold Cache **b.** Hot Cache

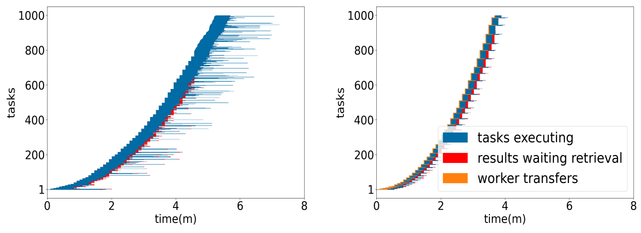
Figure 9: Blast Workflow Cold and Hot Caches
Execution of the BLAST workflow from a worker's perspective. During a cold start, there is substantial overhead due to transferring and staging data. This overhead is removed on subsequent runs.

4.1 Performance Evaluation

Persistent Caching. Persistent caching reduces startup overhead on subsequent executions when objects are cached between workflows. Thus, performing input transfers via the manager or other workers is not necessary. Figure 9 shows the execution the BLAST workflow from Figure 3 first with a cold cluster cache on 100 4-core workers, and then a second time with a hot cache. Startup time dramatically improves.

Shared Mini-Tasks. The addition of features for environment management allows tasks to reuse environments that have already been set up at the worker. This is in opposition to tasks bringing their own environment for each execution. Tasks that share environments can reuse environments left on a worker. With TaskVine, a worker will need to unpack a package once where it can then be used for multiple tasks. This reduces the amount of redundant data that needs to be sent to workers and reduces the execution time for tasks, as a large overhead comes from unpacking the environment. To demonstrate this difference, we ran the same task for both scenarios using 1000 tasks and 50 4-core workers. The task is a minimal task that sleeps for 10 seconds but depends on a 610MB Python package that must be transferred to the worker. Figure 10 shows the difference from tasks reusing an unpacked environment and each tasks unpacking an environment on their own. The environment tarball is transferred to each worker via the manager. Each task using the unpacked environment substantially reduces the time for task execution.

Worker-to-Worker Transfers. Worker-to-Worker transfers were evaluated by measuring the performance of consuming a file at a large number of nodes. Figure 11 compares the distribution of a 200MB file to 500 worker nodes. As a baseline, Figure 11a shows the



a. Independent Tasks **b.** Shared Mini-Tasks

Figure 10: Independent Tasks vs Shared Mini-Tasks
Mini Tasks enable tasks to share staged data, even when it requires some transformation following transfer. **a.** *Each task expands the environment itself as part of its own task definition.* **b.** *Each task shares an expanded environment defined by a shared mini-task.*

result of a worker-to-URL workflow, where a single remote-URL is the source of the input file. Figure 11b shows the importance of the manager in transfer scheduling decisions. Without deliberate usage limits set, the manager will overload a worker and cause performance to suffer. Figure 11c shows the same workflow using worker-to-worker transfers with an imposed concurrent transfer limit of 3, which was found to perform slightly better than two and four. With worker transfers and proper management, the file distribution workflow was complete in approximately half of the original time.

4.2 Example Applications

BLAST [24] is a heuristic DNA text-matching algorithm used within the field of bioinformatics. Given a nucleotide or protein sequence, BLAST is used to search a target database for similar sequences. A typical workflow consists of a large number of query sequences generated by a DNA sequencing device, all being compared against a target database. The primary performance constraint is the distribution of the database replicas and the query software across the system: the more copies available, the higher the sustained query throughput. We constructed a complete workflow in 87 lines of Python, consisting of 2000 tasks on 400 nodes, with the essential elements already shown in Figure 3. Each task receives a query string generated by the manager to search the database. Both the executable and the database are compressed and must first be unpacked by the worker before tasks can be executed. Retrieval and decompression of the remote assets result in significant startup cost, but TaskVine's persistent caching allows these resources to be safely and persistently shared for repeated workflows. Figure 9a shows an execution of the BLAST workflow from the worker's perspective. During a cold start, roughly a quarter of the total execution time is dominated by transferring the necessary assets and staging them. 9b shows the workflow in a subsequent run. Caching these objects on-site minimizes the startup cost.

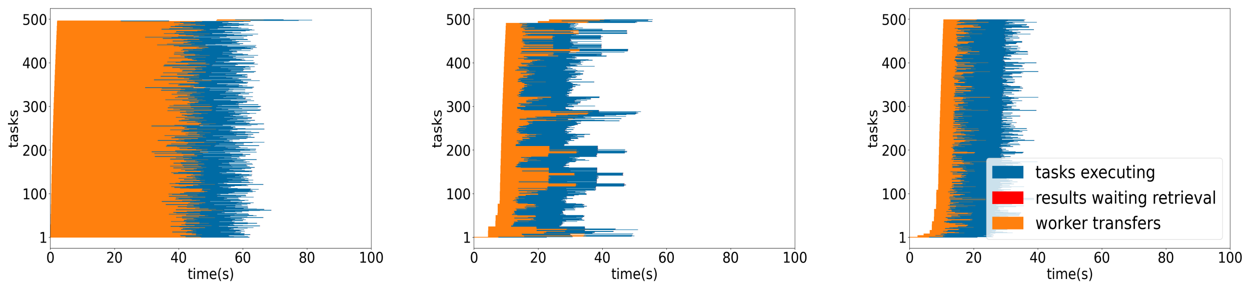
Conclusion: Persistent caching via content-addressable cache names, substantially reduces startup costs for applications with large shared data assets.

TopEFT [8] is a data analysis application in High Energy Physics used to process particle data from the CMS detector at the CERN Large Hadron Collider (LHC), which accelerates protons to nearly the speed of light, producing 40 million collisions per second. The analysis workflows for TopEFT process billions of collision events to calculate and summarize the relevant physics properties. A TopEFT workflow is defined by the datasets of collision events to be analyzed, *preprocessor functions* that collect metadata from the datasets, *processor functions* that generate partial histograms to summarize properties of subsets of collisions, and *accumulator functions* that merge subsets of partial histograms. TopEFT is written on top of Coffea [31], a framework for physics data analysis in python. Coffea prepares the functions to be executed and dispatches them to an executor for completion. We added TaskVine as an execution module for Coffea in about 1300 lines of Python. TopEFT benefits from the use of local temporary files used for partial histograms that remain at the worker storage to be accumulated rather than transferred back and forth to the filesystem. It also makes use of persistent caching of Python environments. Figure 12a displays the completion of tasks with time in the execution of a typical TopEFT workflow processing 0.31 TB of real data collected from the LHC and 1.4 TB of Monte Carlo simulated data. Figure 12d displays the worker view of the run. The number of available workers increases gradually, due to the shared nature of the cluster. In both figures, notice a stall in execution at the 30-minute mark, showing a shift from processing real collisions to processing simulated collisions, which generally require more resources per subset. The output histograms of accumulations grow exponentially in size, and final accumulations produce files in the order of gigabytes.

To visualize the impact of growing accumulations, Figure 13 displays a side-by-side comparison of two TopEFT runs with about 27K tasks. Figure 13a depicts a run in which all output files are brought back to the manager before accumulation. The repeated transfer of large results bottlenecks the system, especially near the end of execution where we observe a delay in data retrieval. The run displayed in Figure 13b takes advantage of TaskVine to keep histograms as ephemeral temporary files that do not leave the workers. Notice the rapid conclusion of the workflow without delays.

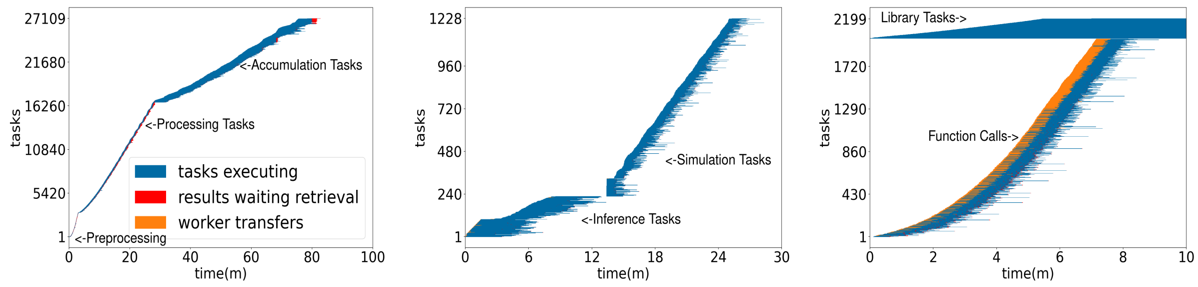
Conclusion: TaskVine's ability to manage data locally within the cluster eliminates the overhead associated with transferring hundreds of gigabytes of data to and from the manager.

ColmenaXTB [35] is a dynamic workflow that combines neural network inferences with molecular dynamics simulations to drive large campaigns of molecular search, and is built on top of the Parsl[7] workflow system. This workflow consists of 228 inference tasks and 1000 simulation tasks, and each task in the workflow displays a complex software dependency and requires 301 software packages (aggregated to 1.4 GB of storage when compressed and archived). We wrote a Parsl module (1036 LOCs in Python) to enable execution of Parsl workflows using TaskVine as a backend system. Without changing the top level workflow, this allows Parsl to exploit TaskVine's worker-to-worker transfer feature to efficiently distribute software packages and share deployments on worker nodes. Figures 12b and 12e displays tasks and workers through the workflow execution, respectively. Notice in Figure 12e that only a small amount of workers receive the tarball of software dependencies from the shared file system in the beginning. They then efficiently distribute copies of the tarball to other workers in the system (3 transfers/worker at any given time), thus reducing the

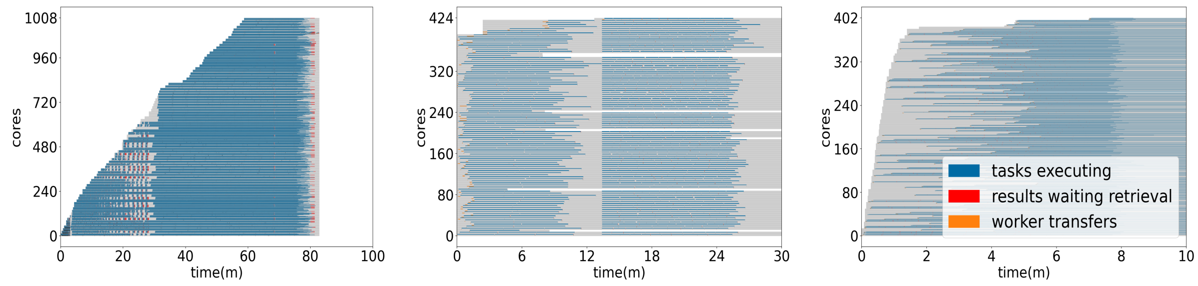


a. Worker-URL **b.** Worker-to-Worker Without Supervision **c.** Worker-to-Worker limited by Manager

Figure 11: Comparison of Transfer Methods for Common Data
a. *500 tasks each independently download input data from a URL.* ***b.*** *Worker-to-Worker transfers are utilized, yet the manager does not limit concurrent transfers.* ***c.*** *With a concurrent transfer limit of 3 per source, an equitable division of bandwidth is achieved.*



a. TopEFT Tasks **b.** Colmena-XTB Tasks **c.** BGD Tasks



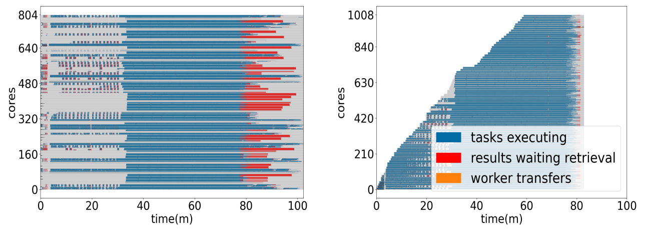
d. TopEFT Workers **e.** Colmena-XTB Workers **f.** BGD Workers

Figure 12: Taskvine Application Evaluation
Three example workflow runs. The top three graphs show activity of each task, sorted by start time. Each row shows the interval of time in which that task was executing. The bottom row of three graphs shows the same execution from the perspective of the workers. Each row indicates that activity of that worker over time. Dark blue indicates a task running, orange indicates data transfer, and light gray indicates an idle worker.

number of queries for the software tarball from the workers to the shared file system from 108 to 3 (the remaining 105 are transfers between workers) without and with worker-to-worker transfers.

Conclusion: TaskVine's worker-to-worker transfer feature substantially reduces the load on the shared file system.

BGD. Batch gradient descent (BGD) is an algorithm which is commonly used to optimize machine learning models during training. The algorithm consists of computing the error of a model on the entire input and adjusting the weights of the model accordingly for a number of iterations. Running many different instances of BGD with different initial models can improve the final error. We have created a BGD workflow in 163 lines of Python using the TaskVine API. This workflow runs 2000 BGD tasks to minimize final model error, and exploits serverless functions to reduce total task overhead throughout the workflow. Each task must create an environment, initialize Python, and resolve imports before actually running BGD. Instead of paying this startup cost once per task, using a Library containing the BGD function allows us to only pay the startup cost once per worker. To start this LibraryTask, the worker requires the high level algorithm code in addition to an environment tarball that is 89 MB. MiniTasks are used to deploy the environment for the Library Instance at the worker. After Library setup, each FunctionCall task can then run BDG on a randomized model with little overhead. Each FunctionCall task takes roughly 50-100 seconds to



a. Shared Storage **b.** In-Cluster Storage

Figure 13: Comparison of TopEFT Execution Modes
a. *Executed on shared storage, resulting in under-utilization at startup and completion.* **b.** *Using shared storage on TaskVine.*

complete. The TaskVine workflow creates 2000 BDG FunctionCall tasks, and then begins installing Libraries on 200 workers. FunctionCall tasks can be sent to workers as soon as their LibraryTask is deployed. This is seen in Figure 12c, as the LibraryTasks are being deployed on the top section of the graph, with the FunctionCall tasks growing from the bottom upwards. The exponential increase in FunctionCall throughput from minute 0 to 5 is due to workers finishing the deployment of their LibraryTask and beginning to run FunctionCalls. After minute 5 is reached, almost all workers have their Libraries deployed, so the FunctionCall throughput of the workflow has peaked as seen by the slope of the task view. This can also be seen in Figure 12f, where almost all workers are running the Library and a FunctionCall by minute 5. This illustrates how the serverless model decreases overall workflow overhead as workers can run many FunctionCalls per LibraryInstance.

Conclusion: The serverless model in TaskVine can improve workflow task throughput by changing task overheads to be performed once per worker instead of once per task.

5 RELATED WORK

Batch and Workflow Systems: HTCondor[32], Slurm[38], LSF[20], and AGE[4] all offer a clean and efficient way to manage shared computing resources on clusters, but lack the ability to manage persistent storage on cluster nodes. TaskVine cooperates with these systems by functioning as an overlay that can run within an existing batch system and manage local storage on behalf of the user. Workflow systems like Kepler[23], Nextflow[13], Pegasus[26], and Galaxy[1] allow direct management of tasks and workers through the use of static DAGs specified in advance of execution. More dynamic workflow systems such as Parsl[7] and Dask[28] instead focus on the problem of distributing tasks to clusters on-the-fly. Parsl targets compute-intensive applications with support for elastic and scalable computation and usability by providing an intuitive Pythonic interface. Dask supports data transfers between workers on demand, but only for data that are results of previous computations. Modern software packages can easily amount to GBs of data, and if unmanaged, will put a large load and bandwidth pressure on external data servers and/or shared file systems. In contrast, TaskVine transfers and distributes the I/O pressure from external data servers to the workers and the local clusters' network through a myriad of techniques, including worker storage management, data-aware scheduling, and general data transfers on-demand between workers.

Startup Time: A number of contemporary works tackle the problem of minimization of startup/re-invocation time of tasks, containers, or function invocations by speeding up the costly I/O operations. Burst buffers provide a high-bandwidth storage platform where checkpoints and output data may be written, so the computation may resume at the node and the data may be asynchronously transferred to the slower parallel file system at the same time [9]. [37] [33] [27] extend the focus of burst buffers to include input data dependencies to expedite job startup. [34] [16] performs an evaluation of BitTorrent on HPC clusters for the purpose of distributing shared libraries among workers and finds that the performance of BitTorrent on an HPC system is affected by issues related to timeouts, as well as the unmanaged nature of the protocol itself. Our work proposes a combination of peer-to-peer file distribution and a data-aware scheduler to avoid contention.

Data Management: Many research groups have mapped out common problems that arise when managing data on systems used for scientific computing. [39] discusses the common conventions for managing and loading package dependencies across HPC systems and presents Shrinkwrap, a tool for resolving such dependencies. [12] documents the need to retrieve and stage data onto the necessary resources for computation. Parallel file systems such as Lustre[10], GPFS[29], and PVFS[30] have been provided as dedicated solutions to the I/O bottleneck problem. I/O forwarding is another approach to the bottleneck problem which restricts I/O operations on compute nodes and forwards them to dedicated I/O nodes[3]. TaskVine avoids bottlenecks by minimizing data movement and exploiting the cluster's bandwidth and storage capacity.

6 FUTURE WORK

Several avenues of future work are possible with TaskVine. In some preliminary work, we have prototyped how Parsl [7] and Dask [28] workflows can execute using TaskVine by simply mapping each high-level task into one low-level TaskVine task, which allows these systems to take advantage of efficient data distribution. But more performance gains are possible when there is a high degree of similarity in the code and data needs that can be distributed once and then invoked multiple times. Future work will explore the automatic transformation of these workflow models into serverless-style computations. A second challenge is the efficiency of scheduling at very large scales. Dynamic workflows that consist of millions of short-running tasks must not only make high quality placement decisions, but also reach those decisions quickly in order to run efficiently: at even one millisecond per task, it would still take a thousand seconds to dispatch a million tasks. This places a fundamental tension between scheduling tasks to necessary data and simply making placements as quickly as possible.

7 CONCLUSIONS

TaskVine is a workflow execution system that seeks to manage data efficiently by exploiting the memory, storage, and network capabilities of cluster nodes. To maintain performance, TaskVine monitors and enforces storage usage and network bandwidth. Experimental results show the improved startup capabilities in high throughput genomics, high energy physics, molecular dynamics and machine learning.

AVAILABILITY

TaskVine is open source software available at: `http://ccl.cse.nd.edu/software/taskvine`.

ACKNOWLEDGMENTS

This work was supported in part by NSF grant OCI-1931348.

REFERENCES

[1] Enis Afgan, Dannon Baker, Bérénice Batut, Marius Van Den Beek, Dave Bouvier, Martin Čech, John Chilton, Dave Clements, Nate Coraor, Björn A Grüning, et al. 2018. The Galaxy platform for accessible, reproducible and collaborative biomedical analyses: 2018 update. *Nucleic acids research* 46, W1 (2018), W537–W544.

[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 *Workshop on Scalable Workflow Enactment Engines and Technologies (SWEET) at ACM SIGMOD*. doi: 10.1145/2443416.2443417.

[3] Nawab Ali, Philip Carns, Kamil Iskra, Dries Kimpe, Samuel Lang, Robert Latham, Robert Ross, Lee Ward, and Ponnuswamy Sadayappan. 2009. Scalable I/O forwarding framework for high-performance computing systems. In *2009 IEEE International Conference on Cluster Computing and Workshops*. IEEE, 1–10.

[4] Altair. [n. d.]. Altair Grid Engine. https://altair.com/grid-engine. Accessed: 2023-03-24.

[5] Ilkay Altintas, Chad Berkley, Efrat Jaeger, Matthew Jones, Bertram Ludäscher, and Stephen Mock. 2004. Kepler: An Extensible System for Design and Execution of Scientific Workflows. In *Proceedings of the International Conference on Scientific and Statistical Database Management, SSDBM*, Vol. 16. 423 – 424. https://doi.org/10.1109/SSDBM.2004.44

[6] Inc Amazon.com. [n. d.]. Amazon Lambda. https://aws.amazon.com/lambda/

[7] 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

[8] Aashwin Basnet, Kenneth Bloom, Florencia Canelli, Sergio Sanchez Cruz, Jose Enrique Palencia Cortezon, Juan Rodrigo González Fernández, Andrea Trapote Fernandez, Reza Goldouzian, Barbara Alvarez Gonzalez, Michael Hildreth, Kevin Lannon, John Lawrence, Sascha Pascal Liechti, Christopher Edward Mcgrady, Kelci Mohrman, Hannah Nelson, Benjamin Tovar, Yuyi Wan, Andrew Wightman, Brian Winer, Furong Yan, Brent R. Yates, Henry Yockey, and Mateusz Zarucki. 2021. TopEFT/topcoffea: TopCoffea 0.1. https://doi.org/10.5281/zenodo.5258003. https://doi.org/10.5281/zenodo.5258003
Source code: https://github.com/TopEFT/topcoffea.

[9] Lei Cao, Bradley W. Settlemyer, and John Bent. 2017. To Share or Not to Share: Comparing Burst Buffer Architectures. In *Proceedings of the 25th High Performance Computing Symposium* (Virginia Beach, Virginia) *(HPC '17)*. Society for Computer Simulation International, San Diego, CA, USA, Article 4, 10 pages.

[10] Sean Cochrane, Ken Kutzer, and L McIntosh. 2009. Solving the HPC I/O bottleneck: Sun™ Lustre™ storage system. *Sun BluePrints™ Online, Sun Microsystems* (2009).

[11] Microsoft Corporation. [n. d.]. Microsoft Azure. https://azure.microsoft.com/en-us

[12] Ewa Deelman and Ann Chervenak. 2008. Data management challenges of data-intensive scientific workflows. In *2008 Eighth IEEE International Symposium on Cluster Computing and the Grid (CCGRID)*. IEEE, 687–692.

[13] 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.

[14] Eelco Dolstra, Merijn De Jonge, Eelco Visser, et al. 2004. Nix: A Safe and Policy-Free System for Software Deployment.. In *LISA*, Vol. 4. 79–92.

[15] Alvise Dorigo, Peter Elmer, Fabrizio Furano, and Andrew Hanushevsky. 2005. XROOTD-A Highly scalable architecture for data access. *WSEAS Transactions on Computers* 1, 4.3 (2005), 348–353.

[16] Matthew G. F. Dosanjh, Patrick G. Bridges, Suzanne M. Kelly, James H. Laros, and Courtenay T. Vaughan. 2014. An Evaluation of BitTorrent's Performance in HPC Environments. In *Proceedings of the 4th International Workshop on Runtime and Operating Systems for Supercomputers* (Munich, Germany) *(ROSS '14)*. Association for Computing Machinery, New York, NY, USA, Article 8, 8 pages. https://doi.org/10.1145/2612262.2612269

[17] Python Software Foundation. 2008. Python Package Index - PyPI. https://pypi.org/.

[18] The Apache Software Foundation. [n. d.]. Apache OpenWhisk. https://openwhisk.apache.org/

[19] Geoffrey C Fox, Vatche Ishakian, Vinod Muthusamy, and Aleksander Slominski. 2017. Status of serverless computing and function-as-a-service (faas) in industry and research. *arXiv preprint arXiv:1708.08028* (2017).

[20] IBM. [n. d.]. Load Sharing Facility. https://www.ibm.com/products/hpc-workload-management. Accessed: 2023-03-24.

[21] Anaconda Inc. 2020. Anaconda Software Distribution. https://docs.anaconda.com/.

[22] Gregory M Kurtzer, Vanessa Sochat, and Michael W Bauer. 2017. Singularity: Scientific containers for mobility of compute. *PloS one* 12, 5 (2017), e0177459.

[23] Bertram Ludäscher, Ilkay Altintas, Chad Berkley, Dan Higgins, Efrat Jaeger, Matthew Jones, Edward A Lee, Jing Tao, and Yang Zhao. 2006. Scientific workflow management and the Kepler system. *Concurrency and computation: Practice and experience* 18, 10 (2006), 1039–1065.

[24] Tom Madden. 2003. The BLAST sequence analysis tool. *The NCBI handbook* (2003).

[25] Dirk Merkel et al. 2014. Docker: lightweight linux containers for consistent development and deployment. *Linux j* 239, 2 (2014), 2.

[26] Suraj Pandey, Karan Vahi, Rafael Ferreira da Silva, and Ewa Deelman. 2018. Event-Based Triggering and Management of Scientific Workflow Ensembles. In *HPCAsia*.

[27] Loïc Pottier, Rafael Ferreira da Silva, Henri Casanova, and Ewa Deelman. 2020. Modeling the Performance of Scientific Workflow Executions on HPC Platforms with Burst Buffers. In *2020 IEEE International Conference on Cluster Computing (CLUSTER)*. 92–103. https://doi.org/10.1109/CLUSTER49012.2020.00019

[28] Matthew Rocklin. 2015. Dask: Parallel Computation with Blocked algorithms and Task Scheduling. In *SciPy*.

[29] Robert B Ross, Rajeev Thakur, et al. 2000. PVFS: A parallel file system for Linux clusters. In *Proceedings of the 4th annual Linux showcase and conference*. 391–430.

[30] Frank B Schmuck and Roger L Haskin. 2002. GPFS: A Shared-Disk File System for Large Computing Clusters. *FAST* 2, 19 (2002).

[31] Nicholas Smith, Lindsey Gray, Matteo Cremonesi, Bo Jayatilaka, Oliver Gutsche, Allison Hall, Kevin Pedro, Maria Acosta Flechas, Andrew Melo, Stefano Belforte, and Jim Pivarski. 2020. Coffea - Columnar Object Framework For Effective Analysis. *CoRR* abs/2008.12712 (2020). arXiv:2008.12712 https://arxiv.org/abs/2008.12712
Source code: https://github.com/CoffeaTeam/coffea.git.

[32] Douglas Thain, Todd Tannenbaum, and Miron Livny. 2005. Distributed Computing in Practice: The Condor Experience. *Concurrency and Computation: Practice and Experience* 17, 2-4 (2005), 323–356. doi: 10.1002/cpe.v17:2/4.

[33] Sagar Thapaliya, Purushotham Bangalore, Jay Lofstead, Kathryn Mohror, and Adam Moody. 2016. Managing I/O Interference in a Shared Burst Buffer System. In *2016 45th International Conference on Parallel Processing (ICPP)*. 416–425. https://doi.org/10.1109/ICPP.2016.54

[34] Shivaram Venkataraman, Aurojit Panda, Ganesh Ananthanarayanan, Michael J. Franklin, and Ion Stoica. 2014. The Power of Choice in Data-Aware Cluster Scheduling. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation* (Broomfield, CO) *(OSDI'14)*. USENIX Association, USA, 301–316.

[35] Logan Ward. 2021 [Online]. Colmena. ExaLearn and Parsl Teams. Available: https://
colmena.readthedocs.io/en/latest/index.html.

[36] Brent Welch, Marc Unangst, Zainul Abbasi, Garth A Gibson, Brian Mueller, Jason Small, Jim Zelenka, and Bin Zhou. 2008. Scalable Performance of the Panasas Parallel File System.. In *FAST*, Vol. 8. 1–17.

[37] Orcun Yildiz, Amelie Chi Zhou, and Shadi Ibrahim. 2017. Eley: On the Effectiveness of Burst Buffers for Big Data Processing in HPC Systems. In *2017 IEEE International Conference on Cluster Computing (CLUSTER)*. 87–91. https://doi.org/10.1109/CLUSTER.2017.73

[38] Andy B. Yoo, Morris A. Jette, and Mark Grondona. 2003. SLURM: Simple Linux Utility for Resource Management. In *Job Scheduling Strategies for Parallel Processing*.

[39] Farid Zakaria, Thomas RW Scogland, Todd Gamblin, and Carlos Maltzahn. 2022. Mapping out the HPC dependency chaos. In *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–12.