Reshaping High Energy Physics Applications for Near-Interactive Execution Using TaskVine

Barry Sly-Delgado
University of Notre Dame
South Bend, U.S.A.
bslydelg@nd

Ben Tovar
University of Notre Dame
South Bend, U.S.A.
btovar@nd.edu

Jin Zhou
University of Notre Dame
South Bend, U.S.A.
jzhou24@nd.edu

Douglas Thain
University of Notre Dame
South Bend, U.S.A
dthain@nd.edu

***Abstract*—High energy physics experiments produce petabytes of data annually that must be reduced to gain insight into the laws of nature. Early-stage reduction executes long-running, high-throughput workflows across thousands of nodes spanning multiple facilities to produce shared datasets. Later stages are typically written by individuals or small groups and must be refined and re-run many times for correctness. Reducing iteration times of later stages is key to accelerating discovery. We demonstrate our experience reshaping late-stage analysis applications on thousands of nodes. It is not enough merely to increase scale: it is necessary to make changes throughout the stack, including storage systems, data management, task scheduling, and application design. We demonstrate these changes when applied to two analysis applications built on open source data analysis frameworks (Coffea, Dask, TaskVine). We evaluate the performance of the applications on opportunistic campus clusters, showing effective scaling up to 7200 cores, thus producing significant speedup.**

***Index Terms*—Parallel Programming, Data Transfer, Physics Computing, Scientific Computing**

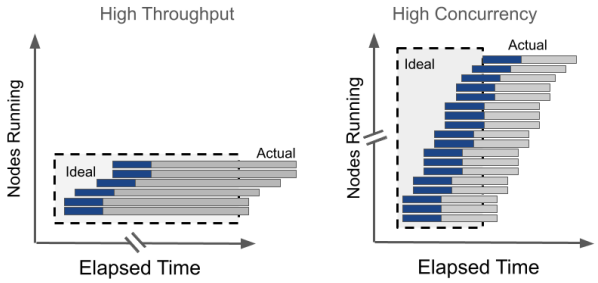


Fig. 1: Application Reshaping
In principle, a high throughput data analysis code can be elastically reshaped by running tasks of 1/10th the size on 10X more nodes. In practice, such reshaping is limited by the overheads of task dispatch, task startup, and data movement. How can we effectively scale up such codes to achieve near-interactive execution times?

I. Introduction

The Compact Muon Solenoid (CMS) experiment at CERN produces petabytes of data annually, collected from particle collisions in the Large Hadron Collider (LHC). The early stages of processing this ”raw” data involve distributed high throughput computing across multiple facilities, making use of hundreds of thousands of cores for weeks to months at a time. [8] These massive workflows result in common ”cooked” datasets that are shared across the CMS collaboration and become the starting point for late-stage data analysis by individuals and small groups. A typical ”small” data analysis carried out by a single investigator might consume TB of data and 10K CPU-hours in order to emit a final publishable result.

Of course, no custom analysis code is correct the first time: it is common to run an analysis many times, troubleshooting and refining the work until a correct outcome is obtained. Reducing the iteration time is critical. In principle, data analysis applications consist mainly of independent tasks and can be reshaped elastically: by running tasks of 1/10th the size on 10X nodes, one should get a 10X speedup. But of course it is not this simple: Figure 1 suggests why: as the data is sliced more finely, tasks are indeed shorter and run on more nodes. But eventually, the fixed costs of each task begin to dominate execution time. These overheads can include things like staging data, initializing software environments, linking and loading libraries, and communicating task state. In order to scale up, applications and frameworks must be designed to minimize or otherwise hide these overheads.

A growing number of high energy physics (HEP) applications are written using high level Python frameworks, in order to take advantage of numeric libraries that are accelerated on GPUs and multi-core CPUs. For example, Coffea [23] encourages analysis codes to be written as a small number of distinct Python functions that can be combined into large graphs and then distributed across a cluster using technologies like Work Queue [32], Dask [20], Parsl [3], Ray [18], and TaskVine [22]. This creates a natural separation of concerns between the application (which defines the analysis), the graph manager (which expresses concurrency), the task scheduler (which makes runtime decisions), and the underlying cluster hardware and distributed filesystem.

Each layer in this stack has an impact on the application’s overall performance. In this paper, we explore improvements to every element of the stack, from the bottom up. First, we replace a legacy HDFS filesystem on spinning disk (high capacity but also high latency) with a modern VAST parallel filesystem based on NVMe storage. While this improves access latency, new hardware alone has minimal impact without further structural improvements. Second, we replace the Work Queue task scheduler with the TaskVine task and data sched-

SC24, November 17-22, 2024, Atlanta, Georgia, USA
979-8-3503-5291-7/24/$31.00 ©2024 IEEE

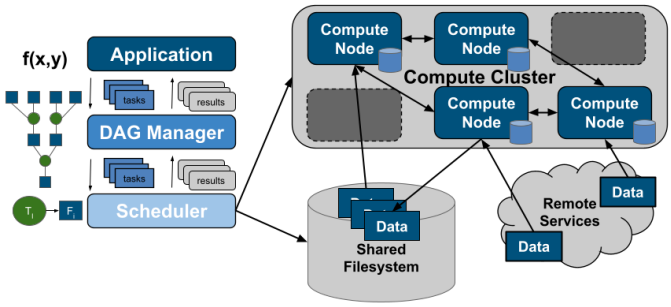


Fig. 2: Architecture of Application Stack
The application defines tasks using the DAG manager, which computes a graph based on the logic defined within the application. The DAG manager then sends tasks to the scheduler, which dispatches tasks to compute nodes in a compute cluster.

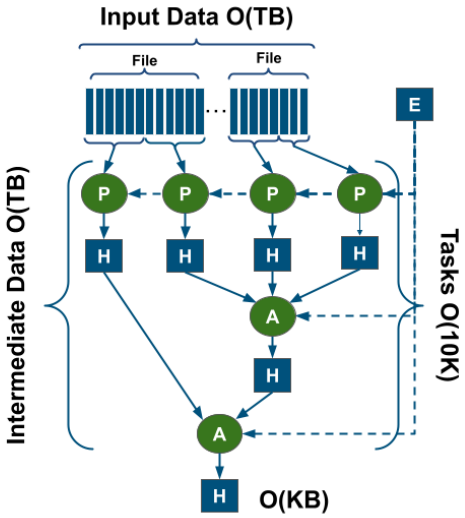


Fig. 3: Generalized Analysis Workflow
The topology of an analysis workflow splits initial input data into "chunks" for processing tasks (P), which produce histograms (H) of relevant physics results. These histograms are then accumulated to form a single histogram via accumulation tasks (A). Each task also requires a software environment (E).

uler in order to exploit node-local storage for intermediate data. Third, we transform the task-oriented execution model into a function-oriented model, which reduces task latency and improves software re-use. Finally, we transform the task graph structure of the application to make more incremental use of distributed storage.

This paper shares our experience of reshaping the entire application stack for production high energy physics applications DV3 [17] and RS-TriPhoton [25]. The combined impact of these hardware and software changes is evaluated on a campus HTCondor cluster [24], demonstrating an improvement in scalability from a few hundred cores up to 7200 cores, effectively reducing application execution time from hours down to minutes. We have contributed our changes within the open source frameworks Coffea, Dask, and TaskVine to enable broader impact upon the high energy physics community.

II. The Application Stack

A growing number of HEP data analysis applications are making use of a multi-layer analysis stack. At the top of the stack exists the **application** itself, which is written in a high level language like Python and expresses user-defined functions applied to abstract objects like structured data sources, distributed arrays, or large arrays. Below the application exists elements needed to distribute and execute tasks. This includes a **DAG manager** which maintains a directed acyclic graph (DAG) composed of many small tasks and data dependencies needed for workflow execution. The DAG manager sits upon a **task scheduler** which dispatches individual tasks to worker nodes for execution. At the lowest level are the systems and facilities used to execute the workflow. This includes specialized data stores, shared filesystems, the compute and storage at worker nodes, and the interfaces which facilitate their interactions.

Figure 2 depicts the overall architecture of this application stack during execution. Tasks flow down through the stack to be eventually executed on workers at the compute nodes. Results subsequently flow up through the stack back to the application, which may trigger the generation of new graphs and tasks to execute.

A. Application Layer

At the application level is the primary logic that defines the analysis computation in terms of high level operations on data sources, data structures, and user defined functions. The specific applications used in this paper are known as DV3 and RS-TriPhoton. DV3 searches collision events to find particle jets that result from decays of the Higgs boson to two bottom quarks and to two gluons. Typical executions of DV3 process datasets on the order of terabytes. RS-TriPhoton searches collision events to find rare signatures of new physics which appear in a three-photon final state, which is the result of a heavy new particle decaying to a photon and a light new particle which then decays to two photons.

Both applications are expressed using Coffea [23], a framework used in HEP to map physics-oriented data formats into familiar objects in the Python data ecosystem. A typical Coffea application refers to a data source in the form of ROOT [5] files which are read into column-oriented data structures. User-defined functions are then applied to these structures in various combinations, usually resulting in an accumulation operation that produces a summary histogram of the result. All of these operations are performed at a bulk abstract level to permit future adaptation to a variety of parallel and distributed systems. Various parameters are available to the user to control the partitioning of the input data as well as the granularity of the processing tasks.

Figure 3 depicts the typical topology for analysis workflows like DV3 and RS-TriPhoton. In general, large amounts of

```python
from ndcctools.taskvine import DaskVine
from coffea.nanoevents import NanoEventsFactory
import hist.dask as hda
import dask

dataset = get_dataset("SingleMu")
events = NanoEventsFactory.from_root(
    dataset,
    permit_dask=True,
    uproot_options={"chunks_per_file": 5}
    metadata={"dataset": "SingleMu"}
).events

hist = (
    hda.Hist.new.Reg(100, 0, 200, name ="met")
    .Double()
    .fill(events.MET.pt)
)

manager = DaskVine(name="my_manager")

hist.compute(
    scheduler=manager.get(),
    peer_transfers=True
    task_mode='function-calls'
    lib_resources={'cores':12, 'slots':12}
    import_modules=[numpy, scipy]
)
```

Fig. 4: Sample Application Code

A simple but complete example of a data analysis application that uses Coffea to fetch data from ROOT, converts it into NanoEvents, constructs a Dask task graph that computes a histogram, and executes it on the cluster using TaskVine.

intermediate data are produced in the form of histograms, requiring large amounts of resources during execution. This has some similarity to Map-Reduce [10] style computation, however more complex programs can easily generate richer, multi-phase graphs. In addition, the reduction phases can often be done hierarchically because the aggregation of histograms is both commutative and associative. Figure 4 shows a simple but complete and executable application expressed using the Coffea framework which consumes HEP data using ROOT and generates a graph of tasks using Dask, and passes it to TaskVine for execution. Both DV3 and RS-TriPhoton are more complex but use this same basic approach.

B. DAG Manager Layer

The high level analysis approach described by the application must be decomposed into a concrete set of tasks to be carried out in a distributed system. These are arranged in a directed acyclic graph (DAG), where each node is a task that consumes specific inputs and produces specific outputs. The DAG naturally expresses constraints upon execution: a task can only be executed once its input dependencies have been created. A DAG manager is responsible for keeping track of the state of each element of the DAG, and dispatching individual tasks and data items as they become ready. A DAG

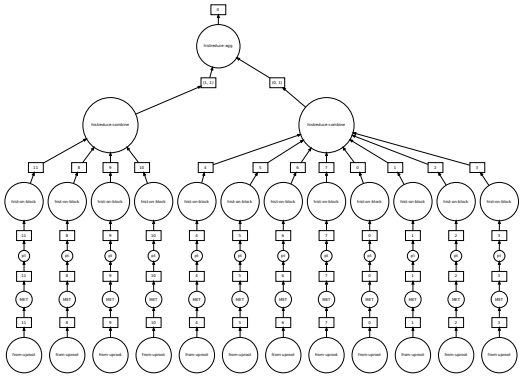


Fig. 5: Sample Task Graph

The task graph generated by Dask from the sample code in Figure 4. The example data set contains 12 files which are processed and then accumulated to a single histogram at top.

structure is naturally flexible and admits a variety of optimizations and deployment decisions across different architectures.

There are a number of frameworks that can express and execute Python workflows in the form of DAGs, such as Parsl [3], Ray [18], and Dask [20]. These HEP applications make use of Dask for its natural coupling with Python data structures. Dask accepts the application's statement of function operations on the underlying data structures, and partitions each element appropriately, internally creating the DAG of Python function calls and data objects. Dask is accompanied by a native task scheduler (Dask.Distributed) that can execute the DAG. However, the DAG can be executed by other schedulers, and (as we show below) we can improve upon both the scalability and performance offered by Dask.Distributed. Figure 5 shows the DAG that is generated by Dask directly from the example code in Figure 4.

C. Scheduler Layer

The job of the scheduler is to dispatch ready tasks and data to specific nodes within a compute cluster. For a standard execution, a given amount of resources (cores, gpus, disk, memory) are allocated for a fixed time period. Utilizing the knowledge of available resources, it is the job of the scheduler to place tasks in a manner for efficient execution. While scheduling tasks, the scheduler must also dispatch data dependencies, and fetch produced results. This is a critical point that is liable to become a bottleneck if not done efficiently.

In this work, we make use of Work Queue [6] as our baseline task scheduler, and then shift to TaskVine [22]. In addition to scheduling tasks, TaskVine also makes use of in-cluster storage and networks to improve data access. Additionally, TaskVine provides multiple execution paradigms for executing remote tasks on compute nodes. With TaskVine, specialized workers are dispatched to compute nodes within a cluster via a batch system. These workers communicate with the centralized TaskVine manager, which dictates all necessary operations regarding data movement, task placement, task execution, and data retention.

D. Storage Layer

Once tasks are dispatched to be executed on a specific compute node, the performance of the task is often constrained by access to needed data. The typical HPC facility provides a shared parallel filesystem mounted on all nodes, but of course physically separate from the compute cluster. In this model, input data is read into the compute node on demand, and outputs written back as produced. If read onto the local disk of a compute node, the performance of those disks affect performance as well. In some cases, data consumed by an analysis may be stored outside of the facility, and fetched on demand from remote services, as suggested in Figure 2. Facilities may set up specialized data stores for improved data access speeds as there exists limitations to solely relying on the shared filesystem. These data stores may be configured to improve data access speeds for specific types of applications. Systems such as HDFS [4] provide cheap bulk high throughput data access while parallel filesystems like VAST [9] can provide low-latency data access for applications.

III. Reshaping Limitations

To transition a high energy physics application from long-running to near-interactive, there are various challenges that need to be addressed. Individual data sets exist on the order of terabytes. Relevant data must be fetched and staged into the compute cluster. For large quantities of data, staging data may make up a significant amount of the runtime for a given application. Additionally, as tasks execute, intermediate data is generated which may be even larger than the initial set of data. Facilitating data movement effectively into and throughout the cluster is a necessary step towards application reshaping. Within a given workflow, tasks may share common data dependencies. However, the standard task execution paradigm may reload shared dependencies adding on overhead to task execution. Thus, this paradigm must be improved upon.

A. Data Access

The CMS experiment at CERN generates petabytes of data annually resulting from particle collisions in the LHC. Conventionally, to provide wide access, data is distributed throughout institutions participating in the experiment. These files are typically formatted as ROOT files, a file format in which columnar data is stored. Remote ROOT files, can be accessed through a variety of means, though XRootD [11] is a protocol specialized for accessing specific columns in remote ROOT files. However, accessing files remotely may contribute significant overhead to staging data into the compute cluster. Because of this, we have procured specialized data stores spatially close to the compute nodes within the cluster. From here, data access times are then limited to the quality of disks on the storage nodes as well as the architecture that facilitates reading from the data stores on the nodes.

B. Data Movement

Once initial data has been accessed and staged into the compute cluster, tasks may produce intermediate results which are to be used for subsequent tasks. To facilitate efficient task scheduling, data may be moved to a new site for execution. In the typical HPC facility, tasks may write intermediate results to a designated location within a shared filesystem. However, as the result is to be used again, it will be read again from the shared filesystem at less than optimal speeds, limited by the bandwidth between storage and compute nodes. Furthermore, for intermediate data that is used by many tasks, this may overload a shared filesystem with metadata requests. In the case that there is no access to a shared storage system, it is possible for a centralized scheduler to stream input and output data to and from compute nodes, storing intermediate results at the location of the scheduler. For a centralized scheduler, this puts large strain on the bandwidth between the scheduler and compute nodes. This may also limit the ability of scheduler to dispatch tasks as it is busy transferring data. This also limits application execution to the disk capacity at the location of the scheduler, as the scheduler cannot store more intermediate results than it has capacity for. One way to alleviate this pressure is to utilize the internal bandwidth and storage of the compute nodes within the cluster. Intermediate results can be kept on disk at the compute node and transferred between nodes when needed. However, to implement this files stored on compute disks must be named consistently for accurate results or even successful execution.

C. Task Execution

Once all the data dependencies needed for task execution are present, tasks can be executed on the compute node. Many high energy physics applications are written in Python, including the applications we've evaluated. The Python environment provides physicists with an array of accelerated libraries to perform their analyses. Conveniently, tasks definitions are often user-defined Python functions. Our applications have defined "processor" functions which analyze input data for relevant physics events. Additionally, there are "accumulation" functions which accumulate these relevant results into a final histogram. Conventionally, to distribute tasks, function objects are serialized along with their arguments and sent to a compute node to be executed using a Python wrapper script, which deserializes the function and its arguments for execution. However, using this method poses some limitations to reshaping applications. Mainly, for each execution, the function is serialized and transferred to the the worker node. As tasks may use common functions, serializing a function object multiple times is redundant. Additionally, at each execution, a wrapper script is executed with the Python interpreter. During this process, specified libraries are loaded for each execution. While the Python interpreter may cache the compiled bytecode of these libraries, for each execution they are still to be read from disk. For functions that use a large number of libraries or a few large libraries this method increases startup costs. One way to alleviate this limitation is with the serverless execution paradigm. With serverless, functions are loaded into a persistent process and invoked remotely, reducing invocations of the Python interpreter. This

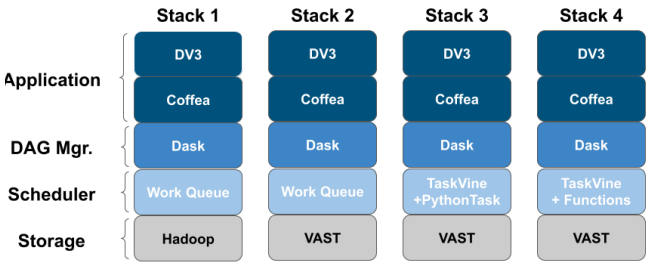


Fig. 6: Application Stack Progression

Incremental improvements to the underlying hardware and software facilitated application reshaping with minimal changes to the application itself. Much of benefit is due to optimizations from using the TaskVine scheduler.

removes the need serialize the function definition per execution and permits loaded libraries to remain in memory.

IV. Structural Improvements

To facilitate application reshaping, we evolved the application stack through changes at the hardware, scheduler, and DAG layers. While changes to the hardware layer provided some improvement, the bulk of improvements come from optimizations in the scheduler layer. Using TaskVine, we are able to retain data at each execution node, enabling peer transfers, and convert conventional tasks into serverless function invocations. Figure 6 shows each stage of the evolution.

Except where noted, each of the improvements in this section was evaluated on a ”standard” run of the DV3 application, consisting of 17,000 tasks consuming 1.2TB of data, running on 200 12-core workers, each with 2.50GHz Intel Xeon CPUs, 96GB RAM, and 108GB of disk. Workers were allocated from a heterogeneous campus HTCondor cluster with opportunistic scheduling, resulting in the preemption of up to 1% of workers in each run. Such preemptions appear as worker ”failures” to the manager, which compensates by replicating data or re-running tasks. Table I shows the performance improvement obtained by each structural change applied incrementally, which we now describe.

Stack	Change	Runtime	Speedup
Stack 1	Original	3545s	1.00x
Stack 2	HDFS → VAST	3378s	1.05x
Stack 3	WQ → TaskVine	730s	4.86x
Stack 4	Tasks → Functions	272s	13.03x

TABLE I: Overall Stack Performance

A. Storage Hardware Improvements

As the applications DV3 and RS-TriPhoton processed large sums of data repeatedly, it was impractical to rely on the wide area XRootD federation to deliver data to each run. Instead, specialized data subsets are maintained at the facility on bulk storage. For a number of years, the CMS group has stored root files on an HDFS [4] cluster consisting of 644TB of spinning disks on commodity hardware nodes, with triple replication. HDFS is specialized towards high-throughput data access rather than low access latency, and as analysis applications became more time sensitive with smaller constituent tasks, latency to access the primary data storage cluster was perceived to be the primary application bottleneck. As the HDFS cluster approached end-of-life, we had the opportunity to move the application over to a new general purpose VAST [9] parallel filesystem procured by our campus HPC center to serve a wide variety of application needs, with 918TB of logical disk space, where 676TB is usable (across 44 15.36GB NVMe SSDs). This filesystem provides much improved latency over HDFS, by virtue of the underlying storage hardware (NVMe drives) as well as a POSIX standard filesystem interface. We expected this dramatic improvement in hardware to provide a notable improvement in application performance, however as Table I notes, the end-to-end impact was quite modest. In its initial state, reading initial data from storage was not (yet) the system bottleneck. Rather, improvements must come from improved task scheduling and intermediate data handling.

B. Scheduler Improvements

The original scheduler for DV3 was Work Queue [6], which consists of a ”manager” that distributes tasks to individual ”workers” running on compute nodes within the cluster. Workers are submitted to the cluster as jobs in a batch system. We replaced the Work Queue scheduler with TaskVine [22], a scheduler that manages both data distribution and task assignments with the cluster. Like Work Queue, TaskVine consists of a manager coordinating workers. However, TaskVine makes extensive use of the local storage at each worker and the available in-cluster bandwidth to store and move data within the cluster, not relying upon a shared distributed filesystem. This results in the following optimizations to the DV3 application:

Retaining Data. As data is produced within the cluster, it is retained on the local storage of each node by the worker process. Retaining data on each worker provides opportunities for the manager to make informed decisions regarding task placement. Additionally, subsequent reads to cached data on the worker’s local disk is typically faster than reading from the shared filesystem at large scale. The manager maintains a mapping of the location of each file within the cluster. Using this mapping, tasks can be scheduled where data dependencies are already available, reducing the need for unnecessary data movement. To implement this, files must be named consistently. That is, files cannot be staged into a worker’s cache simply with their names as they appear to the application or shared filesystem. To name files consistently, TaskVine makes use of file metadata and content to derive a unique ”cachename” for each file in the system. Such cachenames can refer to single files, or to directory hierarchies treated as atomic units. Instead of writing intermediate results to a location outside the compute cluster, or streaming the data back to the manager, intermediate results are retained on the worker nodes and only moved when necessary.

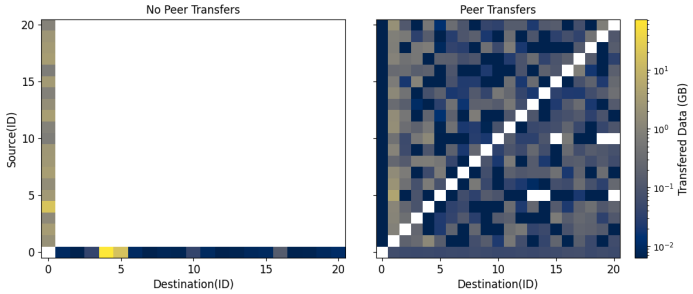


Fig. 7: Data Transfer Comparison

A comparison between executions of DV3 with and without peer transfers (first 20 workers). Each plot displays the total data transferred between nodes during execution. DV3 executed with Work Queue (Left), which only transfers between the manager (node 0) and the workers. Execution of DV3 (Right) with TaskVine, which allows peer transfers between workers.

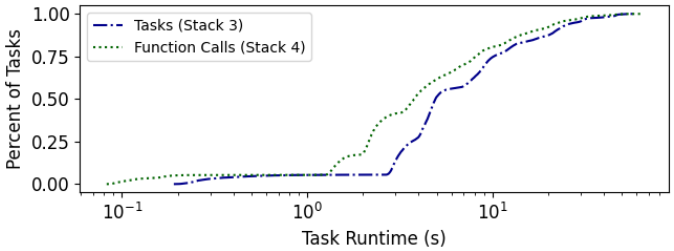


Fig. 8: Standard Tasks vs Function Calls

A comparison of execution modes on runs of DV3. Traditional tasks require the invocation of a Python interpreter for each task. Serverless execution mode keeps the code and libraries resident in memory for multiple executions, which reduces per-task overheads. This has the most significant impact on short-running tasks. (Note log scale)

Peer Transfers. While moving tasks to data is the preferred mode, it may still be necessary to move or replicate files between workers in order to meet the data dependencies of each task. Instead of writing data back to the shared parallel filesystem, TaskVine opts to directly transfer data between worker nodes. To facilitate this, during task placement, the manger will instruct a worker to retrieve a data dependency from a peer. A peer transfer is done asynchronously as to not inhibit task execution. Additionally, the manager manages the number of concurrent peer transfers that a worker may perform, so that uncontrolled peer transfers do not create network contention for frequently used files.

Transferring intermediate data between workers takes advantage of in-cluster disk and bandwidth. Utilizing distributed disk space relieves pressure on the manager and shared filesystem, which would otherwise become a bottleneck for applications that generate large amounts of intermediate data. Stack 2 is primarily constrained by data transfer between the manager and the workers.

To show the advantage of utilizing distributed bandwidth and memory we executed the standard DV3 configuration with the (old) Work Queue scheduler and the (new) TaskVine scheduler making use of peer transfers. Figure 7 shows a heatmap of data transfer between pairs of nodes in each case. When using Work Queue, all data transfer is between the manager (node 0) and each of the workers individually. Upwards of 40GB is transmitted to each worker. When using TaskVine and peer transfers, the maximum amount of data transferred between any two nodes tops off at around 4GB, dramatically relieving the manager (and shared filesystem) of this responsibility.

Serverless Execution. In Python based frameworks such as Dask, Ray, and Parsl, tasks are typically represented as individual user-defined functions. A commonplace practice to distribute tasks to worker nodes involves serializing a function along with its arguments and delivering these files to a worker node, where a wrapper deserializes and executes the function along with the provided arguments. In this mode, there is substantial overhead from transferring serialized functions, reading them from disk, invoking the Python interpreter, and loading any needed libraries

To reduce this overhead, TaskVine offers a serverless task execution mode, in which the application defines a **LibraryTask** containing the function code and any fixed data. This becomes a persistent task which receives invocations to specified functions inside the LibraryTask. Each invocation is known as a **FunctionCall** which contains only the name of the target function and the unique arguments. Upon receiving a function invocation, the library forks a child process to contain the desired invocation. Multiple invocations can run concurrently in order to exploit the total capability of the worker node. This mode dramatically reduces the overhead of task invocation and eliminates distribution of duplicate state.

To evaluate the improvement from serverless execution, we compared the standard task execution mode (Standard Tasks) with the serverless mode (Function Calls) on the standard DV3 workload (DV3 Large). Figure 8 shows the difference in the distribution of task execution times. A majority of tasks have execution times between 1s and 10s (with some outliers on either side). Because the workload consists of 17000 small tasks, task overhead is substantial, and serverless execution has a dramatic effect on overall completion time. Can it be improved further?

Import Hoisting. As described so far, each FunctionCall invocation would find it necessary to import libraries particular to that function at every invocation. This can have a non-trivial cost due to the large number of metadata operations needed to search and traverse the local filesystem. While Python ordinarily caches the bytecode of imported libraries, the fact that the function runs in a child process means the imported library would be lost after each invocation. To reduce this effect, TaskVine provides the ability to ”hoist” import statements inside the FunctionCall to the preamble of the LibraryTask, so that libraries are loaded once per library task rather than per function invocation, execution, reducing the overhead further. Figure 9 shows the structure of import

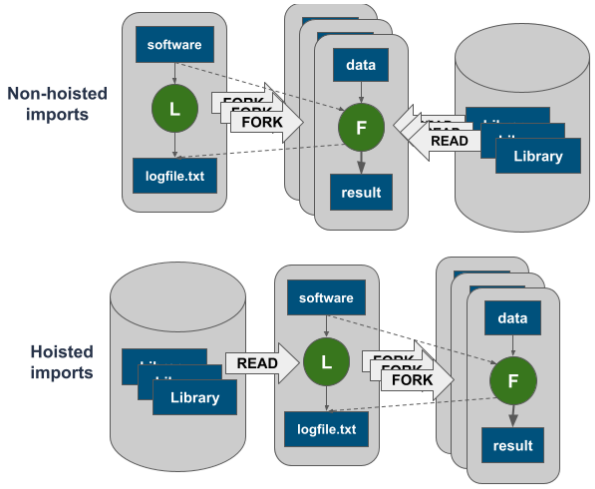


Fig. 9: Import Hoisting

When Imports are hoisted, necessary libraries are only read from disk once per LibraryTask instance. Conversely, if imports are not hoisted, each FunctionTask instance would read libraries from disk. For larger imports, hoisting imports can reduce task execution overhead.

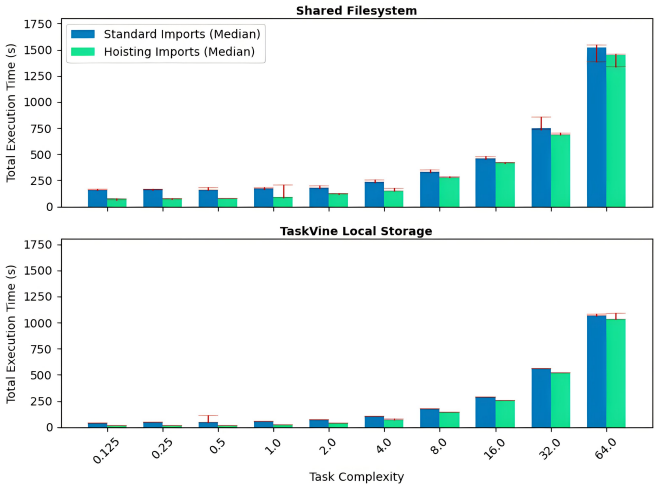


Fig. 10: Import Hoisting Comparison

Execution time of 15K function calls with and without import hoisting. (Top) Import libraries stored in shared filesystem. (Bottom) Import libraries stored in TaskVine local storage. Overall, import hoisting has the most significant effect on small functions.

hoisting within TaskVine.

To evaluate the performance of import hoisting we executed a workflow containing 15,000 independent serverless tasks (function calls) with and without hoisting `import numpy`, comparing TaskVine local storage and the VAST shared filesystem, separately. Each configuration is executed on a set of 16 32-core workers. Additionally, we artificially scale the execution time of a single function from roughly 0.1 seconds to about 35 seconds, which corresponds linearly to a complexity range from 0.125 to 64. Essentially, both filesystems operate on similar workflow patterns, with the TaskVine local storage slightly outperforming the VAST shared filesystem. This improved performance is attributed to localizing library metadata searches to the local disk of each node, rather than traversing the network for the shared filesystem. With import hoisting, each library task on a worker will initially load all of the specified dependencies from the filesystem. Without import hoisting each function call will load the required libraries individually. From Figure 10, we see that significant speedup in shorter running fine-grained tasks. With longer running tasks this speedup is less significant.

C. Application and DAG Improvements

Scheduler improvements are easily incorporated with minimal or no changes to the application itself, as each application uses Coffea, which in turn creates a Dask graph of the resulting workflow. Upon creation of the graph, results can be obtained by calling *compute(x)* for a given task x within the graph. To connect the two layers, we constructed the DaskVine module, which converts the nodes of a Dask graph into task and file submissions to the TaskVine scheduler. A variety of arguments that enable DAG optimizations can then be given to the DaskVine module.

Figure 11 gives an example of this sort of DAG modification. The original RS-TriPhoton application performed a single node reduction that compiled results from all branches of the application in a single task. This required that all input files be transferred to the same node at once, in many cases, overflowing the available local storage and extended overall runtime. Figure 11(a) shows the individual storage consumption of every worker in the system. Each line shows the consumption at a single worker, with arrows indicating reduction operations, and Xs indicating failures. It can be seen that all workers quickly grow to about 200GB of cache usage, but then a few outliers rapidly grow even higher to 700GB or more, and result in the failure and preemption of the worker, and delays in completing the workflow until 6000s. The solution is to modify the DAG as shown in Figure 11(b) so that the reduction is performed as a binary tree, dramatically reducing the quantity of storage needed by each node. In this case, the storage consumption of workers is both reduced and made more uniform, allowing the analysis to succeed and complete in about 1000s.

V. End to End Evaluation

We now take a closer look the end-to-end performance previously summarized in Table I, including improvements made to the application stack throughout its evolution along with the scalability of our improved stack including comparisons to the Dask native Dask.Distributed task scheduler. Here we consider a number of size variations on the DV3 and RS-TriPhoton applications, summarized in Table II.

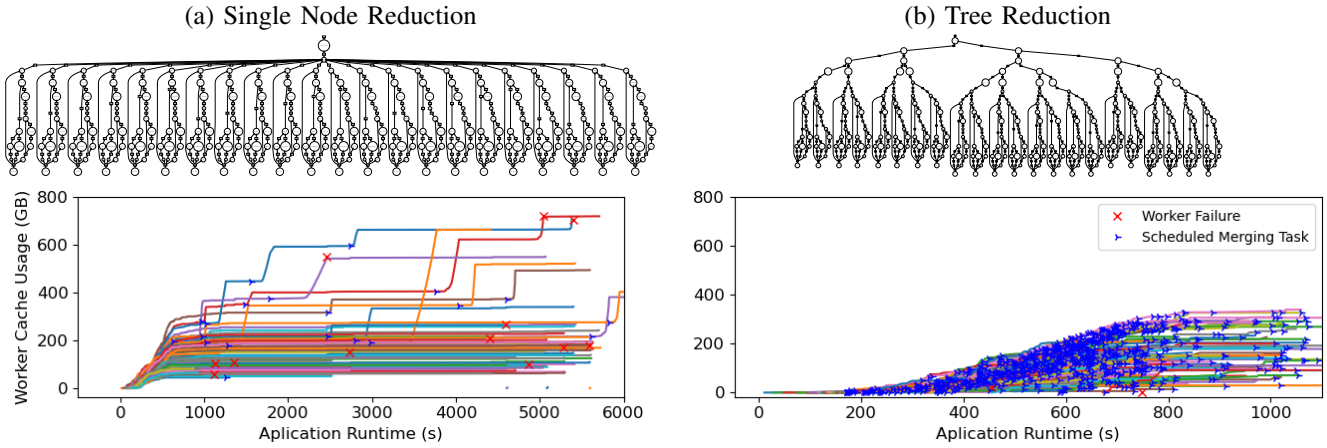


Fig. 11: RS-TriPhoton Reduction Graph Modification

When reducing a graph, hierarchical reduction provides better performance, as a single worker does not become a bottleneck within a cluster. (Left) A single dataset, of 20, is reduced via a single task, causing worker cache utilization to spike and runtime to increase. (right) a dataset is reduced gradually, with improved application runtime and distributed cache utilization.

Application	Input Data Size	Tasks
DV3-Small	25GB	900
DV3-Medium	200GB	1,400
DV3-Large	**1.2TB**	**17,000**
DV3-Huge	1.2TB	185,000
RS-TriPhoton	500GB	4,000

TABLE II: Application Workflow Sizes

A. Application Stack Evolution

Incremental evolution of the hardware and software components of the application stack has allowed us to produce significant gains in speedup during execution. To evaluate these improvements we executed the ”standard” DV3 configuration (DV3-Large in Table II), on each application stack, progressing with improvements to our current stack. Each stack is executed on a set of 200 12-core workers submitted to a compute cluster via HTCondor, where each worker is allocated 98GB of memory and 108GB of disk space.

Transition from Stack 1 to 2 improves upon the storage layer, moving from HDFS to VAST. From Stack 2 to 3, the scheduler layer is improved by incorporating peer transfers to application execution via the move from Work Queue to TaskVine. Lastly, transitioning from Stack 3 to 4, changed task execution paradigms, exchanging the traditional task based execution paradigm to a serverless execution paradigm.

Figure 12 shows a timeline of the first 300 seconds of each stack. (Recall that Stacks 1-3 run much longer than 300s, while Stack 4 completes in 272s.) The top graph shows the number concurrent running tasks, while the bottom graph shows the number of waiting tasks as the workflow executes. Several aspects jump out: Stack 1 sustains high concurrency initially (precisely because its tasks are longer running) but suffers a very long tail during accumulation, with concurrency stabilizing to around 100 tasks. Stack 3 shows oscillating behavior because dispatched tasks complete faster than the next round can be dispatched. This is resolved in Stack 4 which can dispatch and complete function calls much faster than conventional tasks.

Note that improvements of the storage hardware layer (stack 1 to 2) does not produce a significant speedup. Much of the total speedup is attributed towards improvements made within the scheduler. Mainly, the utilization of a worker’s local disk, peer transfers, and serverless task execution. With these improvements, data access speeds are improved by utilizing local disk, data movement costs are distributed among the worker nodes within the cluster, and startup costs are reduced for individual task executions.

To better understand these differences, we measure the performance of DV3-Large on Stack 3 and 4 when utilizing 20 and 200 12-core workers, totalling to 240 and 2400 cores respectively, with the same resource allocations from before. Figure 13 depicts the execution of tasks across workers during executions of Stacks 3 and 4. Colored bars indicate when a task (or function) is actively running on each specific worker. Stack 3 (top) effectively keeps 20 workers busy, but is unable to dispatch and collect tasks fast enough to keep 200 workers consistently working. In contrast, Stack 4 (bottom) is marginally faster than Stack 3 at 20 workers, but much more effective at keeping 200 workers busy, because of the lower overhead of collecting and dispatching function invocations. In this mode, the scheduler is able to distribute task to larger number of workers concurrently, as tasks are now comparatively closer grouped during execution. Because this is a reduction workflow, concurrency drops as the workflow progresses.

B. Scaling Up Workflows

As the bulk of improvements for workflow reshaping were made possible via the scheduler layer, we also compare TaskVine scheduler and workers to Dask.Distributed scheduler and workers, using the same application and underlying

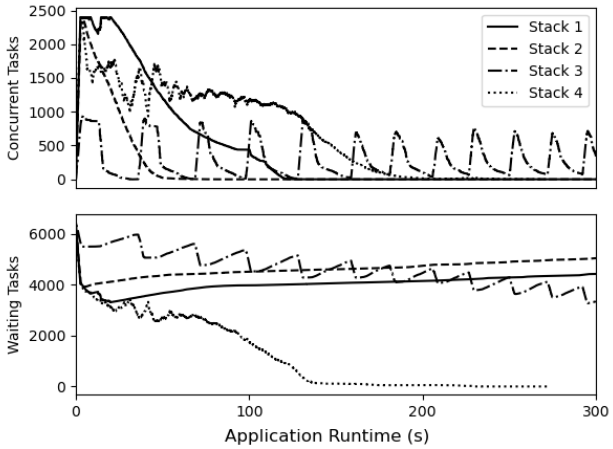


Fig. 12: DV3-Large Stack Evolution
A comparison of workflow execution timeline of DV3-Large as the application stack has progressed. Each execution contains 17,000 tasks. (Top) The number of concurrently executing tasks during application runtime. (Bottom) The number of tasks waiting to be scheduled.

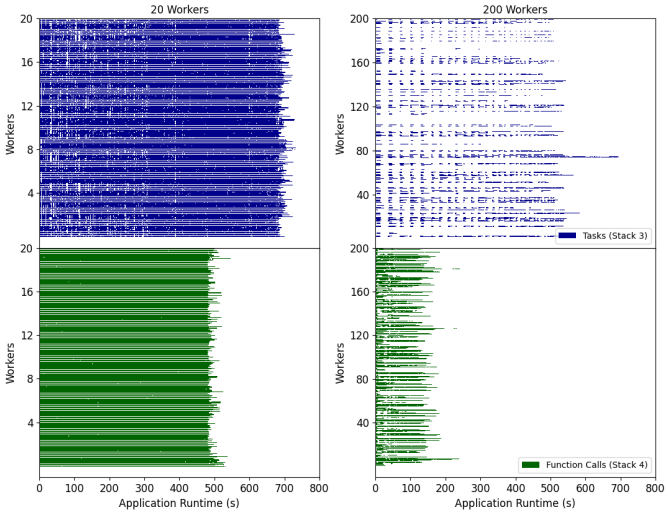


Fig. 13: DV3-Large Stack 3 (Tasks) vs Stack 4 (Functions)
A detailed comparison of the execution behavior of DV3-Large using Stack 3 and Stack 4. Each bar shows where a given worker is actively executing a task or a function. (Top) Stack 3 uses independent tasks, and gains little benefit in scaling from 20 to 200 workers. (Bottom) Stack 4 uses serverless execution resulting in lower overhead and substantial benefit in scaling from 20 to 200 workers.

hardware. These configurations include DV3-Small and DV3-Medium as shown in Table II, which produce workflows with input data sizes of 25GB and 200GB respectively and are scaled from 60 to 300 cores. Figure 14(a) shows that both TaskVine and Dask.Distributed have similar behavior at small scales, however TaskVine completes execution in about 1/2 the time as we approach 300 cores.

There are several structural differences that account for this performance. In TaskVine, each worker manages an entire 12-core compute node, supervising 12 concurrent tasks with a single shared file cache. While Dask.Distributed can be configured in this way, the result is 12 threads competing for a single global interpreter lock, which effectively results in the use of only one core. Instead, it is necessary to run twelve single-core Dask workers that share nothing. In TaskVine, this is not a concern because individual tasks or function calls are processes forked from a common parent.

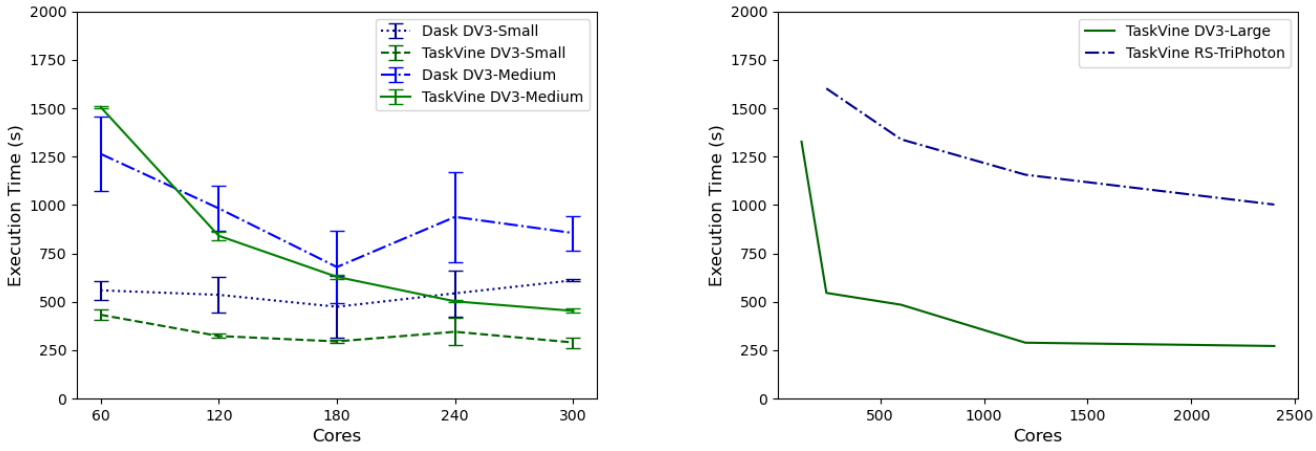
Next, we scale instances of DV3-Large (17K tasks and 1.2TB data) and RS-TriPhoton (4K tasks and 500GB data) from 120 to 2400 cores. RS-TriPhoton has larger memory and disk requirements, and workers in that case are allocated 700GB disk and 200GB of RAM. (Note that Dask.Distributed is unable to execute these workflows at this scale, and consistently fails with a combination of worker and application crashes and hangs.) Figure 14(b) shows that DV3-Large achieves peak performance at 1200 cores, while RS-TriPhoton continues to see small but non-linear gains up to 2400 cores.

Finally, we demonstrate the largest configuration of DV3-Huge, consisting of 185K tasks utilizing the same 1.2TB dataset from the DV3-Large, but is comprised of 185K tasks performing more extensive computation on the same data. This is executed on 600 12-core workers allocated on opportunistic resources. Stack 4 successfully peaks and maintains 7200 cores until available concurrency drops in the reduction stage. What would previously have taken days is completed in less than one hour.

VI. Related Work

Distributed Filesystems - Some of the improvements made that facilitated application reshaping related to improving the hardware in which datasets are stored on. Importantly, the distributed filesystem in which users interact with impacts performance as well. Along with hardware improvements, a change to the distributed filesystem from HDFS to VAST facilitated application reshaping, as HDFS focused on high-throughput data access rather than low-latency. However, The optimal distributed filesystem of choice may be application specific. Other distributed filesystems include AFS [15], NFS [21], Panasas [28], and CEPH [27], which provide general purpose filesystems used in a variety of settings. Though, as the results have shown, initial data access from a shared filesystem only provides a modest improvement to overall application execution.

DAG Based Workflows - DAG based workflows allow users to create distributed applications using familiar syntax. Typically, this results in the formation of a DAG, where each node of the graph is a user-defined task. A variety of frameworks allow for the creation of DAGs such as Dask [20], Parsl [3], Ray [18], Makeflow [2], and Pegasus [19].



(a) (b)

Fig. 14: Scaling

:*The scaling of applications shown in Table II (excluding DV3-Huge). (Left) scaling comparison between TaskVine and Dask's native Dask distributed scheduler. TaskVine produces improved scalability and execution times. (Right) Scaling of standard configurations of DV3 (DV3-Large) and RS-TriPhoton.*

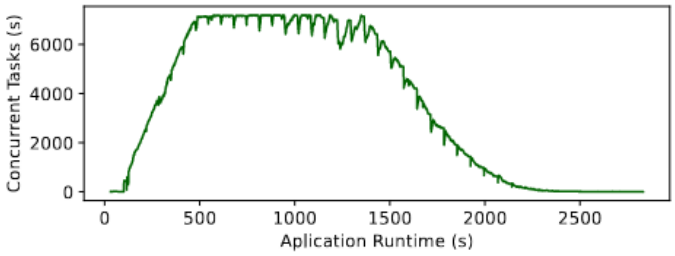


Fig. 15: DV3-Huge

Full scale DV3 analysis. The generated workflow contains 185,000 tasks with 10,000 initial executable tasks from the start. TaskVine maintains high concurrency during the duration of the execution until the reduction of the graph.

Users can then express their distributed applications via the provided API, which may be language specific. This specific paradigm of application creation is opposed to application creation using MPI [26]. In turn, this allows for DAG-based optimizations where domain scientists need not worry about optimizations in regard to load balancing, task placement, and data management as it is then handled by a designated scheduler. However, frameworks like Swift/T [31] can translate user code, into MPI code, which may provide improvement on HPC resources. Distinctively, TaskVine provides an interface, DaskVine, which allows it to be used for HEP workflows, which have a more rigid application stack that uses Dask as the DAG manager. Thus, the incorporation of other workflow managements systems is a challenge for HEP applications. While Map-Reduce [10] defines workflows using map and reduce functions, which are applicable to general HEP processing and accumulation functions. Though, using this framework would involve entirely refactoring current HEP applications. Additionally, TaskVine provides flexibility with the ability to execute non-restricted DAGs as opposed to Map-Reduce. This kind of DAG restriction is present in other runtimes such as Twister[12] DryadLINQ [13], and Spark [30].

Workflow Executors - A workflow executor or scheduler, distributes tasks within a workflow to sites for execution. These sites include allocated resources on compute clusters, grids, or the cloud. During execution, a static workflow may be given in its entirety to the scheduler, while individual tasks within a dynamic workflow may be given to the scheduler incrementally. Frameworks that provide workflow executors include TaskVine[22], Work Queue[7], Dask [20], Ray [18], and Parsl [3]. The scheduler facilitates task placement, and may facilitate data movement as well, with different schedulers having the ability to handle dynamic or static workflows. These schedulers typically communicate to "workers" running on allocated resources submitted via a batch system such as Slurm [29], Flux [1], SGE [14], or LSF [16]. To utilize different schedulers for HEP applications, an interface such as DaskVine must be used to facilitate communication between the native Dask DAG manager and scheduler. Otherwise, the entire application may need to be refactored to utilize a specific scheduler. Our results have shown that the TaskVine scheduler can outperform Dask's native Dask.Distributed scheduler by utilizing TaskVine's built-in capabilities.

VII. Conclusions

The optimizations present in TaskVine facilitate application reshaping by effectively managing data across computation sites. Namely, by implementing peer transfers across compute nodes, alleviating stress on the manager and shared filesystem and allowing for increased time to schedule and distribute

tasks. This spreads responsibility for data movement across nodes in a cluster. Additionally, TaskVine reduces startup overhead by incorporating serverless task execution, reducing the amount of invocations of Python interpreter by invoking functions within a persistent library task process. Furthermore, the overhead cost of importing libraries needed for task execution is reduced via import hoisting, which allows for users to provide libraries which are to be imported within the preamble of a library task. With improvements to the hardware, filesystem, and workflow scheduler, we facilitated application reshaping on DV3, a high energy physics analysis application, producing a speedup of 13X.

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

REFERENCES

[1] D. H. Ahn, N. Bass, A. Chu, J. Garlick, M. Grondona, S. Herbein, H. I. Ingólfsson, J. Koning, T. Patki, T. R. Scogland, et al. Flux: Overcoming scheduling challenges for exascale workflows. *Future Generation Computer Systems*, 110:202–213, 2020.

[2] M. Albrecht, P. Donnelly, P. Bui, and D. Thain. 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*, SWEET '12, New York, NY, USA, 2012. Association for Computing Machinery.

[3] Y. Babuji. Parsl: Pervasive parallel programming in python. HPDC '19, New York, NY, USA, 2019. Association for Computing Machinery.

[4] D. Borthakur. The hadoop distributed file system: Architecture and design. *Hadoop Project Website*, 11(2007):21, 2007.

[5] R. Brun and F. Rademakers. Root—an object oriented data analysis framework. *Nuclear instruments and methods in physics research section A: accelerators, spectrometers, detectors and associated equipment*, 389(1-2):81–86, 1997.

[6] P. Bui, D. Rajan, B. Abdul-Wahid, J. Izaguirre, and D. Thain. Work Queue + Python: A Framework For Scalable Scientific Ensemble Applications. In *Workshop on Python for High Performance and Scientific Computing (PyHPC) at the ACM/IEEE International Conference for High Performance Computing, Networking, Storage, and Analysis (Supercomputing)* , 2011.

[7] P. Bui, D. Rajan, B. Abdul-Wahid, J. Izaguirre, and D. Thain. Work Queue + Python: A Framework For Scalable Scientific Ensemble Applications. In *Workshop on Python for High Performance and Scientific Computing (PyHPC) at the ACM/IEEE International Conference for High Performance Computing, Networking, Storage, and Analysis (Supercomputing)* , 2011.

[8] S. Chatrchyan, E. de Wolf, P. Van Mechelen, et al. The cms experiment at the cern lhc. *Journal of instrumentation.-Bristol, 2006, currens*, 3:S08004, 2008.

[9] V. Data. Data platform built for deep learning and ai. https://vastdata.com/.

[10] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[11] A. Dorigo, P. Elmer, F. Furano, and A. Hanushevsky. Xrootd-a highly scalable architecture for data access. *WSEAS Transactions on Computers*, 1(4.3):348–353, 2005.

[12] J. Ekanayake, H. Li, B. Zhang, T. Gunarathne, S.-H. Bae, J. Qiu, and G. Fox. Twister: a runtime for iterative mapreduce. In *Proceedings of the 19th ACM international symposium on high performance distributed computing*, pages 810–818, 2010.

[13] Y. Y. M. I. D. Fetterly, M. Budiu, Ú. Erlingsson, and P. K. G. J. Currey. Dryadlinq: A system for general-purpose distributed data-parallel computing using a high-level language. *Proc. LSDS-IR*, 8, 2009.

[14] W. Gentzsch. Sun grid engine: Towards creating a compute power grid. In *Proceedings First IEEE/ACM International Symposium on Cluster Computing and the Grid*, pages 35–36. IEEE, 2001.

[15] J. H. Howard, M. L. Kazar, S. G. Menees, D. A. Nichols, M. Satyanarayanan, R. N. Sidebotham, and M. J. West. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems (TOCS)*, 6(1):51–81, 1988.

[16] IBM. Load sharing facility. https://www.ibm.com/products/hpc-workload-management. Accessed: 2023-03-24.

[17] C. Moore. Dv3. https://github.com/cmoore24-24/hgg/blob/main/processors/dv3_processor.py, 2024.

[18] P. Moritz. Ray: A distributed framework for emerging AI applications. In *OSDI*, 2018.

[19] S. Pandey, K. Vahi, R. F. da Silva, and E. Deelman. Event-based triggering and management of scientific workflow ensembles. In *HPCAsia*, 2018.

[20] M. Rocklin. Dask: Parallel computation with blocked algorithms and task scheduling. In *SciPy*, 2015.

[21] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon. Design and implementation of the sun network filesystem. In *Proceedings of the summer 1985 USENIX conference*, pages 119–130, 1985.

[22] B. Sly-Delgado, T. S. Phung, C. Thomas, D. Simonetti, A. Hennessee, B. Tovar, and D. Thain. TaskVine: Managing In-Cluster Storage for High-Throughput Data Intensive Workflows. In *18th Workshop on Workflows in Support of Large-Scale Science*, 2023.

[23] N. Smith, L. Gray, M. Cremonesi, B. Jayatilaka, O. Gutsche, A. Hall, K. Pedro, M. A. Flechas, A. Melo, S. Belforte, and J. Pivarski. Coffea - Columnar Object Framework For Effective Analysis. *CoRR*, abs/2008.12712, 2020.
Source code: https://github.com/CoffeaTeam/coffea.git.

[24] D. Thain, T. Tannenbaum, and M. Livny. Condor and the Grid. In F. Berman, A. Hey, and G. Fox, editors, *Grid Computing: Making the Global Infrastructure a Reality*. John Wiley, 2003. isbn: 0-470-85319-0.

[25] A. Townsend. Rs-triphoton. https://github.com/atownse2/skim-test/blob/main/skim.py, 2024.

[26] D. W. Walker and J. J. Dongarra. Mpi: a standard message passing interface. *Supercomputer*, 12:56–68, 1996.

[27] S. Weil, S. A. Brandt, E. L. Miller, D. D. Long, and C. Maltzahn. Ceph: A scalable, high-performance distributed file system. In *Proceedings of the 7th Conference on Operating Systems Design and Implementation (OSDI'06)*, pages 307–320, 2006.

[28] B. Welch, M. Unangst, Z. Abbasi, G. A. Gibson, B. Mueller, J. Small, J. Zelenka, and B. Zhou. Scalable performance of the panasas parallel file system. In *FAST*, volume 8, pages 1–17, 2008.

[29] A. B. Yoo, M. A. Jette, and M. Grondona. Slurm: Simple linux utility for resource management. In *JSSPP*, 2003.

[30] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster computing with working sets. In *2nd USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 10)*, 2010.

[31] Y. Zhao, J. Dobson, L. Moreau, I. Foster, and M. Wilde. A notation and system for expressing and executing cleanly typed workflows on messy scientific data. In *SIGMOD*, 2005.

[32] C. C. Zheng and D. Thain. Integrating Containers into Workflows: A Case Study Using Makeflow, Work Queue, and Docker. In *Workshop on Virtualization Technologies in Distributed Computing (VTDC)*, 2015. doi: 10.1145/2755979.2755984.

Appendix: Artifact Description/Artifact Evaluation

I. Overview of Contributions and Artifacts

A. Paper's Main Contributions

- C_1 Methods for reshaping high energy physics applications from long-running to near-interactive using TaskVine.
- C_2 Evaluation of serverless task execution for physics application reshaping.
- C_3 Evaluation of utilizing peer transfers to for physics application reshaping.

B. Computational Artifacts

A_1 https://doi.org/10.5281/zenodo.12583626

Artifact ID	Contributions Supported	Related Paper Elements
A_1	C_1, C_2, C_3,	Figures 7-8, 11-15 Table 1

II. Artifact Identification

A. Relation To Contributions

The artifact provided, (A_1), includes plotting tools used to generate the original figures in the paper along with the logs used to generate these figures. Additionally, the artifact includes scaled-down experiments used to ensure and verify the reproducibility of the artifact. The provided experiments use a reduced data set to scale down execution time and needed resources. A set of experiments can executed either locally or on distributed resources. The resources provided within the artifact substantiate our paper contributions by providing the original logs of the experiments used in the paper and scaled-down experiments for replication.

B. Expected Results

The scaled-down experiments within the artifact should replicate the significant contributions shown in the paper. Specifically, for figures 7-8 and Figures 11 - 15. As the artifact produces replicated figures for the ones shown in the paper, the expected results are as follows:

Figure 7 - The replication of figure 7 should depict two runs of the application DV3 where the plot on the left is depicting no data transfers between peer workers while the plot on the right will depict more data transfer among peer workers, where the max amount of data transferred between any two peers is smaller when peer transfers are not enabled.

Figure 8 - The replication of figure 8 produces a plot depicting the runtime of tasks during an execution of DV3 in two modes, Tasks and Function Calls. Here, the tasks executed in Function Calls mode should be faster.

Figure 11 - The replication of figure 11 should depict a scaled-down version of the application RS-Triphoton, depicting worker cache usage over the application's runtime. Specifically, this is executed in two modes, single and binary graph reeducation. Binary graph reduction should use less distributed disk space and execute faster.

Figure 12 - The replication of Figure 12 depicts executions of DV3 on various stack configurations as mentioned in the paper. The replicated figure depicts 2 graphs, consisting of pertinent information during runtime. Specifically, concurrent tasks and waiting tasks. This replication contains 3 lines instead of the 4 shown in the paper. The reason being that stack 1 and 2 use different hardware that may not be available for individuals executing artifact experiments. Instead, stacks 1 and 2 are depicted as a single entity. The expected results for this replicated figure should show stack 4 having high concurrent tasks while having low waiting tasks. Stack 3 should have less concurrency resulting in more waiting tasks. The combination of stacks 1 and 2 (noted as stack 2) should have even less concurrency and even more waiting tasks.

Figure 13 - The replication of figure 13 should depict the task stream of executions of DV3 on a set of workers for stacks 3 and 4. Here, the task stream for stack 3 should be more segmented than the stream of stack 4.

Figure 14 - The replication of figure 14 should depict the executions of DV3 and RS-Triphoton at different scales (cores) with workflow executors TaskVine and dask.distributed. Here TaskVine should perform better than dask.distributed at each scale Additionally, scaling is shown for DV3 and RS-TriPhoton.

Figure 15 - The replication for figure 15 should depict the amount of concurrent tasks during an execution of DV3. Here, Concurrency should be maintained high for the majority of the duration of the execution.

C. Estimated Runtime

There are two main experiments presented in the artifact, DV3 and RSTriPhoton. The experiment for DV3 runs in 4 configurations where each configuration is run 3 times. Execution for each configuration is ran at 2 scales. RS-TriPhoton is run in 2 configurations, each 3 times at 3 scales.

1) Artifact Setup: Once downloaded, there is little setup for execution (less than 10 minutes).

2) Artifact Execution: Each execution of DV3 can range from 10 - 15 minutes. For all runs at all scales and configurations, this can take from 4 - 6 hours. Each execution of RS-TriPhoton completes in roughly a minute. This experiment takes roughly 20 minutes.

3) Artifact Analysis: Once execution is complete, the analysis of the reproduced experiments should generate graphs. Graph generation should take less than 10 minutes.

D. Artifact Setup

1) Hardware: If executed on a compute cluster, compute nodes should be equipped with a local disk and have access to a shared filesystem. Additionally, if executed on a a compute cluster, suitable batch schedulers include SLURM, SGE, and HTCondor.

2) Software: The artifact should be executed on a LINUX-based machine. For execution, an installation of Conda $\geq$ 24.4.0 is needed to install software dependencies via a YAML file. Package versions are listed within the YAML file for each experiment's environment.

3) Datasets/Inputs: Datasets are provided within the artifact, where each data set is located under its respective experiment's directory.

4) Installation and deployment: Once Conda is installed and the repository has been downloaded, there are two directories, "paper_figures" and "experiments". First, uncompress large logs by running the command: **"./uncomporess_large_logs.sh"**. Within the experiments directory, traversing to either directory "DV3" or "RS-TriPhoton", individuals can download necessary software dependencies via the command: **"conda env create –name <ENVIRONMENT_NAME> –file=env.yml"**. Then to generate a tarball needed for environment distribution, activate the environment and run the command:**"poncho_package_create $CONDA_PREFIX <dv3-env|rstri-env>.tar.gz"**. Then run the experiment by executing the command: **"./run.sh"**. Once each experiment is complete, a user can run: "**./gen_graphs.sh**" located in the experiments directory. This will produce the replicated figures mentioned above. Notably, Values are to be entered manually for replications of figure 14. these values can be retrieved from the logs for dask.distributed executions and generated using command: **"vine_plot.py LOG_NAME –compute-time"** for TaskVine executions.

E. Artifact Evaluation

The steps to evaluate the artifact are as follows: once the repo has been downloaded and setup has been completed, execute the experiments located in the directories **experiments/DV3** and **experiments/RS-TriPhoton** as shown in the Artifact Setup (installation and deployment) section above. This produces the logs used for artifact replication. Then generate the replicated figure by executing "**./gen_graphs.sh**". For each experiment, 4 core workers are used to reduce the scale. Additionally, 64GB of memory and 100GB of disk space is allocated for execution. Each configuration, for each scale during an experiment is repeated 3 times. Most of these parameters can be modified within the run script. Notably, the size of the datasets for DV3 and RS-TriPhoton are 1.2 and .5 gigabytes respectively. This is to reduce replication compute time.

F. Artifact Analysis

For the figures, information is taken from logs generated on each execution. penitent information is computed from these logs to generate for displays. For each execution, logs will be outputted into the **experiments/logs** directory. To generate the displays the logs are parsed and relevant information is computed. information such as task placement, data transfer events, and task run times are computed from these logs. There are various events that are accumulated to produce a figure. For example, figure 12 is generated via computing task placement and execution duration from the logs.