Fusion and Reshaping of Hidden Task Graphs for High Energy Physics Applications

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

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

***Abstract*—Workflow management systems allow users to create large-scale distributed scientific applications by translating user-provided code or specifications into a task graph. In the high energy physics (HEP) domain, users provide functions that represent tasks in a broader directed acyclic graph (DAG). However, the automatically constructed topology of the DAG may inhibit the performance of the application due to poor task granularity, increased overhead, reduced concurrency, and decreased fault tolerance. The task graph itself is often not visible to the user, as it is generated automatically. This work provides the methodology for restructuring workflows via task fusion and reshaping. Task fusion combines tasks into a single executable entity, reducing overhead. Task reshaping leverages algebraic properties of a task's function such as variable arity, commutativity, and associativity. This is done using labels for safe restructuring of the task graph. This is implemented into the composition of the frameworks Dask and TaskVine. We then show improvement with real-world HEP applications in terms of runtime, task overhead, and fault-tolerance, improving end-to-end latency by at least 30%.**

I. Introduction

Workflow management systems (WMS) enable users to construct large-scale distributed scientific applications. These applications are commonly represented as task graphs, or directed acyclic graphs (DAGs), where nodes represent tasks and edges represent dependencies. In many systems, the resulting DAG topology is implicitly generated during execution and is not directly exposed to users. Within high energy physics (HEP), Dask [18] is widely used with the Coffea framework [23] to provide Python-based abstractions for scalable analysis. User-defined functions serve as task definitions within a workflow that is synthesized and executed at runtime. The resulting graph may include both computational tasks and data dependencies representing intermediate inputs and outputs.

Automatically generated DAGs may contain inefficient structures due to suboptimal task granularity, scheduling overhead, limited concurrency, and reduced fault tolerance. Fine-grained tasks increase scheduling and serialization costs, while coarse-grained tasks reduce scheduling flexibility by occupying resources for longer periods. Additionally, graph topology influences data movement: fan-outs may require intermediate data transfers between workers, while large fan-ins can create resource contention and increase execution time. Figure 1 illustrates two workflows that produce the same output. The left workflow performs a flat accumulation after processing, while the right workflow applies task fusion before execution and reshapes the reduction into a hierarchical structure. These transformations reduce unnecessary task boundaries while exposing additional parallelism. Constructing optimized workflows remains challenging because task granularity and safe graph transformations are not generally known at the application level. In particular, reshaping reductions requires knowledge of algebraic properties such as variable arity, associativity, and commutativity. In this work, we introduce an intermediate layer within the application stack that automatically restructures workflows without requiring users to reason about underlying DAG topology.

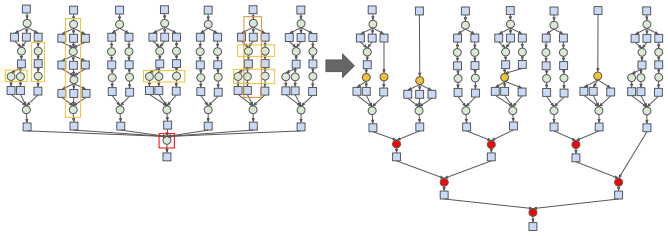


Fig. 1: Graph Restructuring
Transformation of a task graph through fusion and reshaping. Circles represent tasks and squares represent data dependencies. Fused tasks are grouped within yellow boundaries, while the final fan-in is reshaped into a hierarchical reduction within the red boundary.

This work provides the following contributions:

1) We characterize inefficiencies in automatically generated HEP task graphs, including granularity imbalance, scheduling overhead, and data movement costs.
2) We introduce a task graph rewriting framework using algebraic annotations to enable safe, automatic workflow fusion and reshaping.
3) We demonstrate cross-layer optimization within a Dask–TaskVine execution stack, enabling workflow restructuring without changes to user-level application code.

We implement our approach within the Dask and TaskVine execution stack and evaluate it using CMS-based [5] HEP applications including RsTriPhoton [25], DV5 [13], and ttBar [15]. Our results demonstrate improvements in runtime, task

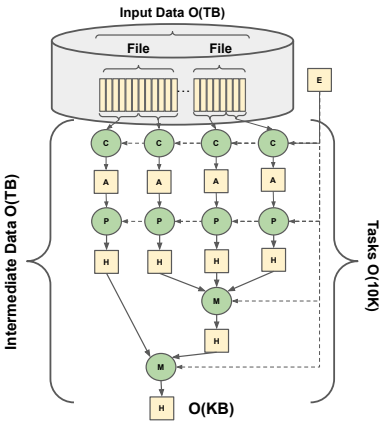


Fig. 2: High Energy Physics Workflow
Simplified HEP workflow. Input data is converted to a columnar format (C), processed (P), and merged (M) into a final result. Tasks may share a common software environment (E).

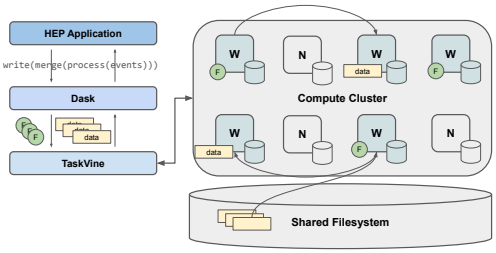


Fig. 3: WMS Application Stack
Workflow stack for HEP analyses. User code generates a Dask task graph, which is scheduled by TaskVine and executed across distributed worker nodes.

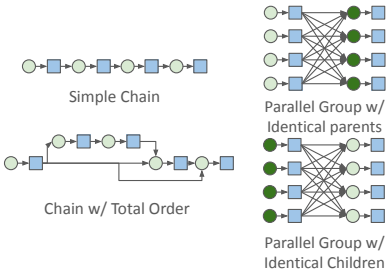


Fig. 4: Task Fusion
*Fusion patterns considered. **Vertical** chains are fused when they form a simple chain or total order. **Horizontal** groups are fused when they share identical parent or child sets.*

overhead, and fault tolerance, achieving at least a 30% reduction in end-to-end latency.

II. Structure

High energy physics analyses from the CMS experiment at the Large Hadron Collider (LHC) process large volumes of data. Early processing stages reduce petabytes of detector data to dataset-scale collections on the order of terabytes, which are then analyzed by individual research groups. These analyses are commonly implemented using the Coffea framework, which provides Python abstractions for scalable columnar analysis built on NumPy [10] and Awkward Array [16].

We evaluate three applications: RsTriPhoton, which searches for rare three-photon signatures of new physics; DV5, which applies kinematic cuts before energy correlation function (ECF) analysis; and ttBar, which studies top-quark pair production. Figure 2 illustrates the workflow structure. ROOT [3] input files are converted to a columnar representation via conversion tasks (C), processed in chunks by processing tasks (P), and produce intermediate outputs (H), which are merged (M) into a single Parquet [26] file per dataset. Chunk size is determined by a user-defined step size. Users implement the conversion, processing, and merge functions, while the workflow management system constructs the resulting DAG for distributed execution.

A. Workflow Management Systems

Workflow management systems translate user-defined code or workflow specifications into distributed task graphs. Workflows may be expressed declaratively (e.g., YAML) or programmatically (e.g., Python or C++). Systems such as Dask, Ray [14], and Parsl [2] provide different programming models but share a common architecture consisting of an **application**, **DAG manager**, **task scheduler**, and distributed **task executors**. The application defines tasks, the DAG manager constructs and maintains the workflow graph, the scheduler dispatches ready tasks to workers, and completed tasks enable dependent tasks. Figure 4 illustrates this architecture for our HEP workflows, where Dask serves as the DAG manager and TaskVine as the scheduler.

B. Task Fusion

Task fusion combines multiple graph nodes into a single executable task that produces one output. Figure 4 illustrates the fusion patterns considered. We fuse *vertical* chains consisting of single-parent, single-child dependencies, including chains that form a total order despite intermediate branching. We also fuse *horizontal* groups whose tasks share identical parent sets or identical child sets.

C. Task Reshaping

Task reshaping transforms a task into an equivalent subgraph. We focus on reduction tasks with large fan-in, replacing flat reductions with hierarchical reductions. Eligible reductions must support variable arity, associativity, commutativity, and compatible input/output structures.

III. Task Fusion

Task fusion combines multiple workflow tasks into a single executable unit. Different fusion strategies offer distinct trade-

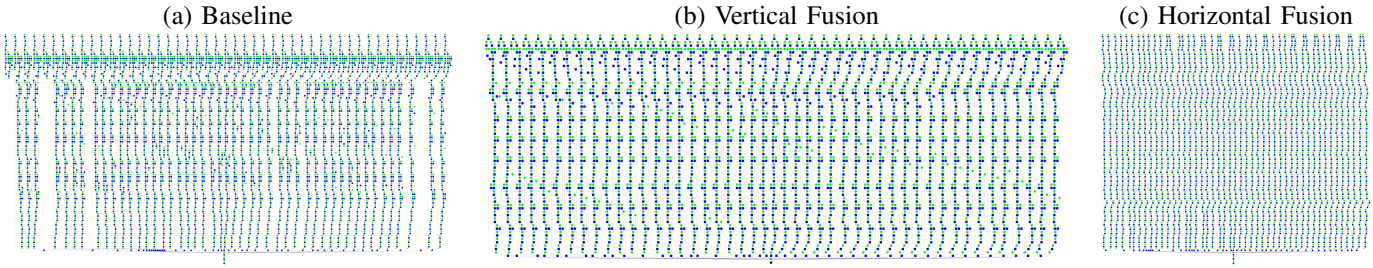


Fig. 5: Fusion Outcomes

A comparison between fusion outcomes on single dataset within RsTriPhoton. (a) No modifications to the task graph are performed. (b) Some subgraphs have been fused vertically, reducing the depth of the task graph. (c) Groups have been merged horizontally, reducing the apparent width of the task graph. These graphs are read from top-down.

offs. At one extreme, no fusion preserves maximum task granularity; at the other, complete fusion collapses the workflow into a single execution unit.

A. Considerations

Distributed workflow execution incurs serialization, data movement, communication, and scheduling overheads. Fusion reduces serialization by combining multiple functions into a single executable with an internal execution order. Vertically fused chains can eliminate intermediate writes by passing values directly between operations, while horizontally fused groups transfer shared inputs once for reuse across all constituent tasks. Fewer tasks also reduce scheduling and coordination overhead.

B. Limitations

Fusion also introduces trade-offs. Vertically fused chains retain intermediate results in memory rather than writing them to disk, increasing memory consumption, particularly for long chains. Longer-running fused tasks also occupy compute resources for extended periods, reducing scheduling flexibility. Horizontally fused groups may duplicate outputs when downstream tasks require only subsets of the results, increasing redundant computation or data movement. Figure 5 illustrates the effects of vertical and horizontal fusion on an RsTriPhoton workflow.

IV. Task Reshaping

Task reshaping transforms a single task into an equivalent subgraph, enabling more efficient execution of workflows with large fan-ins.

A. Considerations

Task execution incurs serialization, scheduling, dependency management, and data movement overheads. For fine-grained tasks these costs can dominate execution time [22]. Flat reductions are particularly susceptible because a single merge task must stage all inputs, increasing data movement, disk utilization, and merge time. Hierarchical reductions partition the merge into smaller tasks, reducing per-task input size while exposing additional merge parallelism. On distributed systems, hierarchical reductions also improve resource utilization. Intermediate data are distributed across workers rather than concentrated on a single node, reducing disk pressure and balancing network traffic. Because multiple merge tasks execute concurrently, reductions continue to exploit available parallelism late in the workflow. Hierarchical reductions also improve resilience on opportunistic resources such as HTCondor, where worker eviction or failure may invalidate intermediate data stored on compute nodes. A flat reduction requires recomputation from its original inputs, whereas a hierarchical reduction introduces additional intermediate results that serve as restart points, reducing recomputation after failures. Figures 6b and 6c illustrate equivalent 23-ary and 3-ary reduction trees. Smaller reduction arities increase the number of restart points and improve resilience during merging. To enable these transformations, we introduce user-defined algebraic annotations as Python decorators. These annotations expose properties such as variable arity, associativity, and commutativity, allowing the workflow manager to safely identify reducible fan-ins during DAG construction. Reshaping therefore becomes a compile-time graph transformation driven by explicit user annotations rather than graph inference.

B. Limitations

Hierarchical reductions increase the number of merge tasks, introducing additional scheduling, dependency management, and data staging overhead. For inexpensive merge operations or small intermediate results, these costs may outweigh the benefits of increased parallelism. They also lengthen the critical path, as results must propagate through multiple merge levels before the final output is produced.

V. Implementation

DaskVine is an intermediate layer within the TaskVine codebase that enables TaskVine to serve as a scheduler when Dask is used for DAG management.

A. Task Fusion

Task fusion is performed within DaskVine between Dask and TaskVine. After generating the initial task graph, DaskVine first applies vertical fusion, merging eligible linear chains until no further fusion is possible. It then applies

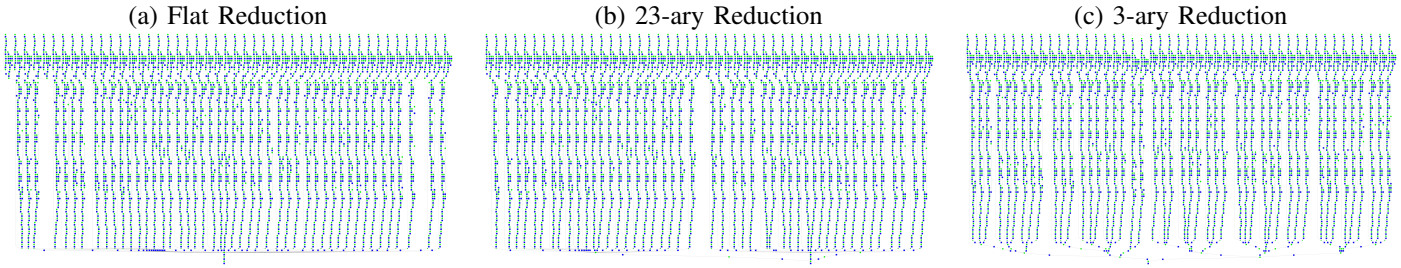


Fig. 6: Reduction Reshaping

Reduction topologies for a single RsTriPhoton dataset. (a) Flat reduction. (b) Hierarchical 23-ary reduction. (c) Hierarchical 3-ary reduction. Smaller reduction arities increase intermediate merge points while producing the same final result. Graphs are shown top-to-bottom.

TABLE I: Performance of evaluated applications.

RsTriPhoton	**# Tasks**	**# Files**	**Total Data (GB)**	**Data Transferred (GB)**	**Runtime (s)**	**Wrkr. Util.**	**Task Util.**
Original	9738	9738	1022	3220	1037	**.87**	.71
Vertical Fusion	6118	6118	807	2500	753	**.87**	**.75**
Horizontal Fusion	7321	7321	963	2653	850	.79	.68
Horizontal + Vertical	**4382**	**4382**	**670**	**2032**	702	.86	.73
Reshaping Only	9780	9780	1080	3250	1004	.86	.71
Reshaping + Fusion	4424	4424	730	2200	**668**	.86	**.75**
DV5	**# Tasks**	**# Files**	**Total Data (GB)**	**Data Transferred (GB)**	**Runtime (s)**	**Wrkr. Util.**	**Task Util.**
Original	318176	318176	1080	1635	2090	.80	.64
Vertical Fusion	199800	199800	853	1269	1646	.81	**.68**
Horizontal Fusion	239200	239200	1017	1347	1860	.81	.67
Horizontal + Vertical	**143200**	**143200**	**708**	**1032**	1535	.79	**.68**
Reshaping Only	318576	318576	1100	1680	2027	**.82**	.65
Reshaping + Fusion	144600	144600	771	1117	**1460**	.81	**.68**
ttBar	**# Tasks**	**# Files**	**Total Data (GB)**	**Data Transferred (GB)**	**Runtime (s)**	**Wrkr. Util.**	**Task Util.**
Original	33440	33440	1502	4137	2193	.86	.70
Vertical Fusion	21070	21070	1237	3656	1537	**.87**	.75
Horizontal Fusion	24280	24280	1330	3640	1754	.80	.70
Horizontal + Vertical	**16032**	**16032**	**1170**	**3413**	1456	.86	.73
Reshaping Only	33478	33478	1538	4070	2039	.85	.71
Reshaping + Fusion	16053	16053	1210	3342	**1358**	.86	**.76**

horizontal fusion to groups of tasks with identical parent or child structure, subject to a constraint that the fused group does not exceed the maximum branch width.

The resulting graph is used for execution. Although horizontal fusion may expose additional vertical fusion opportunities, we do not apply fusion recursively, as doing so can produce a degenerate graph in which each branch becomes a single fused task, substantially reducing concurrency.

B. Task Reshaping

Task reshaping can be performed manually by the user or automatically by the workflow management system. A reduction is eligible for restructuring when its underlying function has variable arity, associativity, and commutativity, and its output structure is compatible with its input structure.

Because these properties are not directly observable by the system, we introduce user-provided semantic annotations through Python decorators. Figure 7 shows an example using the `@merge` decorator. These annotations encode the algebraic properties of reduction functions as explicit labels attached to user-defined code. The annotations are propagated through DAG construction and interpreted by DaskVine before execution, enabling graph restructuring without requiring changes to the underlying workflow.

DaskVine implements a reduction optimizer that reshapes annotated subgraphs. The initial Dask DAG represents a reduction as a single merge task consuming all intermediate results. Given a valid reshaping annotation, DaskVine replaces this flat reduction with a hierarchical reduction tree. For example, the flat reduction in Figure 4a consists of a single merge task with 47 input dependencies. Reshaping replaces this node with a structured set of intermediate merge tasks forming a balanced reduction tree.

This transformation preserves the input-output semantics while changing the execution topology. Because the annotated merge operation is associative, commutative, and variable-arity, the transformation is semantically valid. The resulting DAG is submitted to the scheduler using the same task representation as an unmodified graph, leaving the scheduler agnostic to whether the topology was constructed originally or produced through reshaping.

```
from coffea import NanoEventsFactory
from ndcctools.taskvine import DaskVine
import awkward as ak

@merge
def merge_results(*events):
    return ak.concatenate(events)

def write(events):
    return ak.to_parquet(events, "out")

# construct graph
graph = {}
f = convert(dataset)
f = process(f)
f = dask.delayed(merge_results)(f)
f = dask.delayed(write)(f)
graph[f"dataset_name_write"] = f

#compute results
m = DaskVine()
scheduler = m.get()
f = dask.compute(f, reconstruct=True)
```

Fig. 7: Code Example

*A simplified HEP workflow. The **merge** annotation marks `merge_results` as an associative reduction, allowing the workflow manager to perform task reshaping automatically.*

VI. Evaluation

We evaluate RsTriPhoton, DV5, and ttBar, HEP analyses that skim datasets to extract relevant physics results. Datasets are partitioned into 250,000 events per partition for RsTriPhoton, 50,000 for DV5, and 100,000 for ttBar. We execute workers using UGE and HTCondor on a campus cluster. UGE provides stable resources with longer wait times, while HTCondor provides more resources quickly but with potential eviction.

Each workflow consists of three stages: conversion, processing, and merging. Conversion tasks read ROOT [3] files from low-latency VAST [7] storage near the compute nodes. This differs from HEP workflows that use XRootD [9] to access remote data. For N partitions, conversion produces N read tasks. Processing filters events for relevant physics properties and provides the widest concurrency through many short-running tasks. Finally, merging combines the N results into a single file using either a flat or hierarchical reduction. Hierarchical reductions use a merge size of 3, informed by the task completion results.

A. Main Results

For each application, we execute six configurations three times: (i) original workflow, (ii) vertical fusion, (iii) horizontal fusion, (iv) combined vertical and horizontal fusion, (v) reshaping only, and (vi) combined fusion and reshaping. Graph restructuring takes less than one second for all workflows. We query each resulting DAG for task count and estimate unique intermediate data using prior runs, as shown in the left portion of Table I.

RsTriPhoton and ttBar use two datasets and 10 eight-core UGE nodes with 250,GB SSD storage and 64,GB memory. DV5 uses a single dataset and 15 twelve-core workers with the same storage and memory configuration. All nodes use AMD EPYC 7543 processors at 2.80,GHz. Fusion and reshaping reduce end-to-end latency by 35

B. Data Movement

Fusion retains intermediate data in memory rather than writing it to disk. As shown in Table I, configurations combining horizontal and vertical fusion generate the least unique intermediate data and transfer the least data. TaskVine creates two replicas per intermediate result for fault tolerance, explaining why transferred data can approach 3X the generated data. Fusion with reshaping generally provides the second-best results, as hierarchical reduction introduces additional intermediate tasks.

C. Utilization

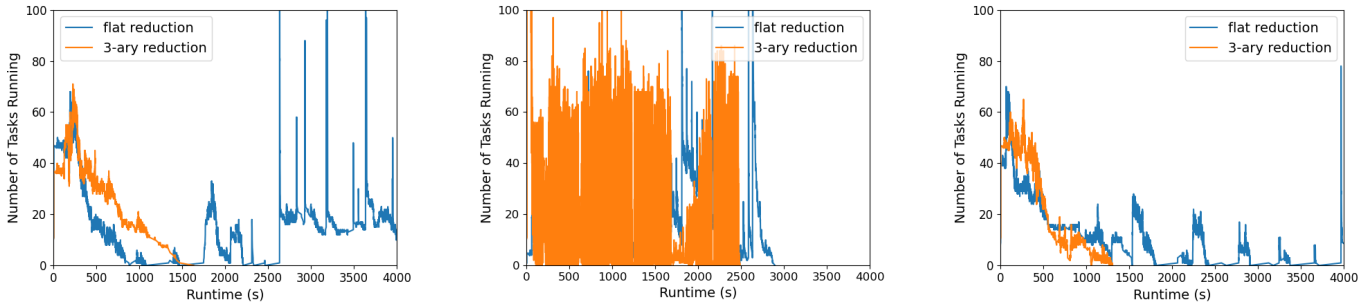
Table I reports two utilization metrics. *Wrkr. Util.* measures worker utilization from the manager's perspective, from task dispatch to completion notification, while *Task Util.* measures core utilization from the worker's perspective, from executable fork to completion. Their difference captures data movement and communication overhead. Fusion generally increases task utilization by moving more work to workers, while worker utilization varies across runs.

D. Fault Tolerance

We evaluate fault tolerance by executing the applications on HTCondor while simulating node loss every minute. We compare hierarchical reduction with merge size 3 against a flat reduction. Figure 8 shows concurrently executing tasks over time. Hierarchical reduction increases runtime relative to the average runtime, but flat reduction produces a substantially larger increase. Each concurrency spike represents *recovery tasks* scheduled after intermediate results are lost.

E. Task Completion

We measure the time required to complete 90, 95, 99, and 100 percent of tasks for RsTriPhoton on a single dataset. Figure 9 compares merge sizes of 2, 3, 4, 5, 6, and a flat reduction, averaged across three runs. Larger merge sizes generally increase completion times by reducing available concurrency. The gap between 99 and 100 percent completion also grows with merge size and is largest for the flat reduction. Although the flat reduction reaches 99 percent completion fastest, its final completion time substantially exceeds that of the hierarchical configurations.



(a) RsTriPhotn (b) DV5 (c) ttBar

Fig. 8: Application Fault Tolerance

Task concurrency during runtime for each application under a controlled failure regime, where one worker is evicted every minute to simulate periodic node loss in a distributed execution environment.

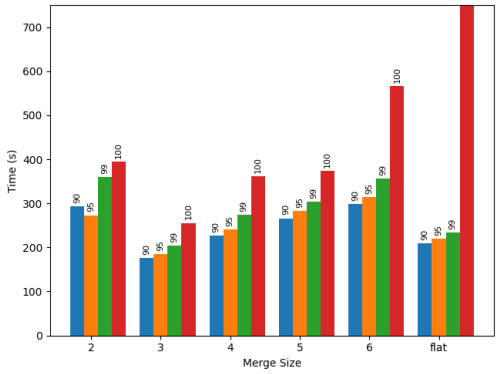


Fig. 9: Task Completion

The time taken to complete N percent of tasks within the RsTriPhoton task graph using different merge sizes from 2 to 6 and a flat reduction. As merge size increases the disparity between completing 99 percent and 100 percent of tasks increases.

VII. Related Work

Workflow Management Systems - Dask, Parsl, and Ray enable users to construct distributed applications from Python. These systems provide graph optimizations such as task deduplication and fusion, but do not generally restructure graphs using algebraic properties of user-defined functions. Our approach introduces this capability within DaskVine and is applicable to other workflow systems. Systems such as Spark [30] and Hadoop [11] similarly optimize map-reduce workflows, but are less commonly used in HEP analysis pipelines.

Storage Systems - TaskVine exploits node-local storage to reduce data movement by retaining intermediate results near computation. This complements shared filesystems such as Panasas [28], GPFS [19], Lustre [6], AFS [12], and NFS [21], as well as burst-buffer systems [27], [17], [29], [4]. Unlike conventional burst buffers, TaskVine uses compute nodes directly as intermediate storage, combining the compute and storage layers. In our evaluation, VAST [7] provides the persistent storage for input ROOT files.

User-Annotated Optimizations - Our approach uses explicit user annotations to expose semantic properties unavailable through static or runtime graph analysis [24]. This resembles compiler optimization [20], where semantic information enables transformations while preserving correctness. Here, annotations allow the DAG optimizer to transform task graphs into more efficient execution plans.

Task Graph Rewriting - Task graph compilation systems such as TensorFlow, XLA, and Apache Spark Catalyst [1] rewrite intermediate representations to improve execution. Unlike these systems, which operate on constrained operator graphs with known semantics, HEP workflow DAGs are heterogeneous and user-defined. Our annotations provide the semantic information required for safe restructuring.

Reduction Tree Optimization - Hierarchical reductions are widely used in parallel and distributed systems to reduce contention and communication overhead, including tree-based MPI reductions [8]. Our work extends this principle to arbitrary user-defined DAGs by making reduction structure an explicit optimization target.

VIII. Conclusion

We present a methodology for task graph restructuring using user-provided annotations that encode algebraic properties of reduction operations. An intermediate layer uses these annotations to combine task fusion and reshaping: fusion reduces DAG overhead by merging task chains and groups, while reshaping converts large fan-ins into hierarchical executions. Across three HEP applications—RsTriPhoton, DV5, and ttBar—combining these techniques reduces end-to-end latency by at least 30

References

[1] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, et al. Spark sql: Relational data processing in spark. In *Proceedings of the 2015 ACM SIGMOD international conference on management of data*, pages 1383–1394, 2015.

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

[3] 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.

[4] L. Cao, B. W. Settlemyer, and J. Bent. To share or not to share: Comparing burst buffer architectures. In *Proceedings of the 25th High Performance Computing Symposium*, HPC '17, San Diego, CA, USA, 2017. Society for Computer Simulation International.

[5] 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.

[6] S. Cochrane, K. Kutzer, and L. McIntosh. Solving the hpc i/o bottleneck: Sun™ lustre™ storage system. *Sun BluePrints™ Online, Sun Microsystems*, 2009.

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

[8] J. J. Dongarra, S. W. Otto, M. Snir, D. Walker, et al. An introduction to the mpi standard. *Communications of the ACM*, 18(11), 1995.

[9] 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.

[10] C. R. Harris, K. J. Millman, S. J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith, R. Kern, M. Picus, S. Hoyer, M. H. van Kerkwijk, M. Brett, A. Haldane, J. F. del Río, M. Wiebe, P. Peterson, P. Gérard-Marchant, K. Sheppard, T. Reddy, W. Weckesser, H. Abbasi, C. Gohlke, and T. E. Oliphant. Array programming with NumPy. *Nature*, 585(7825):357–362, Sept. 2020.

[11] A. Holmes. *Hadoop in practice*. Simon and Schuster, 2014.

[12] J. H. Howard et al. *An overview of the andrew file system*, volume 17. Carnegie Mellon University, Information Technology Center, 1988.

[13] C. Moore. Dv5. https://github.com/cmoore24-24/hgg, 2025.

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

[15] H. Nelson. ttbar. https://github.com/TopEFT/ttbarEFT/tree/coffea2025, 2025.

[16] J. Pivarski, I. Osborne, I. Ifrim, H. Schreiner, A. Hollands, A. Biswas, P. Das, S. Roy Choudhury, N. Smith, M. Goyal, P. Fackeldey, and I. Krommydas. Awkward Array, 10 2018.

[17] L. Pottier, R. F. da Silva, H. Casanova, and E. Deelman. Modeling the performance of scientific workflow executions on hpc platforms with burst buffers. In *2020 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 92–103, 2020.

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

[19] F. B. Schmuck and R. L. Haskin. Gpfs: A shared-disk file system for large computing clusters. *FAST*, 2(19), 2002.

[20] P. B. Schneck. A survey of compiler optimization techniques. In *Proceedings of the ACM annual conference*, pages 106–113, 1973.

[21] S. Shepler, B. Callaghan, D. Robinson, R. Thurlow, C. Beame, M. Eisler, and D. Noveck. Network file system (nfs) version 4 protocol. Technical report, 2003.

[22] B. Sly-Delgado, B. Tovar, J. Zhou, and D. Thain. Reshaping High Energy Physics Applications for Near-Interactive Execution Using TaskVine. In *(to appear) ACM/IEEE Supercomputing*, pages 1–11, 2024.

[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] B. Tovar, B. Lyons, K. Mohrman, B. Sly-Delgado, K. Lannon, and D. Thain. Dynamic Task Shaping for High Throughput Data Analysis Applications in High Energy Physics. In *IEEE International Parallel and Distributed Processing Symposium*, 2022.

[25] A. Townsend. Rstriphoton. https://github.com/atownse2/skim-test/blob/main/skim.py, 2025.

[26] D. Vohra. Apache parquet. In *Practical Hadoop Ecosystem: A Definitive Guide to Hadoop-Related Frameworks and Tools*, pages 325–335. Springer, 2016.

[27] T. Wang, K. Mohror, A. Moody, K. Sato, and W. Yu. An ephemeral burst-buffer file system for scientific applications. In *SC '16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 807–818, 2016.

[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] O. Yildiz, A. C. Zhou, and S. Ibrahim. Eley: On the effectiveness of burst buffers for big data processing in hpc systems. In *2017 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 87–91, 2017.

[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.