DaYu: Optimizing Distributed Scientific Workflows by Decoding Dataflow Semantics and Dynamics

Meng Tang[2,1], Jaime Cernuda[2], Jie Ye[2], Luanzheng Guo[1], Nathan R. Tallent[1], Anthony Kougkas[2], Xian-He Sun[2],
[1]Pacific Northwest National Laboratory, [2]Illinois Institute of Technology
Emails: mtang11, jcernudagarcia, jye@hwak.iit.edu; lenny.guo, tallent@pnnl.gov; akougkas@iit.edu, sun@iit.edu

2024 IEEE International Conference on Cluster Computing (CLUSTER) | 979-8-3503-5871-1/24/$31.00 ©2024 IEEE | DOI: 10.1109/CLUSTER59578.2024.00038

***Abstract*—The combination of ever-growing scientific datasets and distributed workflow complexity creates I/O performance bottlenecks due to data volume, velocity, and variety. Although the increasing use of descriptive data formats (e.g., HDF5, netCDF) helps organize these datasets, it also introduces obscure bottlenecks due to the need to translate high-level operations into file addresses and then into low-level I/O operations. To address this challenge, we introduce DaYu, a method and toolset for analyzing (a) semantic relationships between logical datasets and file addresses, (b) how dataset operations translate into I/O, and (c) the combination across entire workflows. DaYu's analysis and visualization enable the identification of critical bottlenecks and the reasoning about remediation. We describe our methodology and propose optimization guidelines. Evaluation on scientific workflows demonstrates up to a 3.7x performance improvement in I/O time for obscure bottlenecks. The time and storage overhead for DaYu's time-ordered data are typically under 0.2% of runtime and 0.25% of data volume, respectively.**

I. Introduction

Scientific discovery increasingly relies on distributed scientific workflows to orchestrate interconnected tasks, integrating data processing, interpreting experimental results, conducting theoretical analyses, and preparing for future experiments. These workflows span disciplines such as physics, bioinformatics, and material science [1][2]. Tasks within a workflow can range from individual steps within a single application to spanning different applications. Downstream tasks depend on upstream tasks, creating intricate dependencies throughout the workflow [3]. Workflow execution performance is typically limited by I/O bottlenecks since most inter-task data exchanges primarily rely on shared storage resources like Parallel File Systems (PFS) [4], [5].

We focus on the challenges of I/O bottlenecks in workflows that exchange data using increasingly popular descriptive data formats, like Hierarchical Data Format (HDF)[6], that abstract I/O. As shown in Figure 1, these formats (a) provide logical data structures and operations; (b) map the logical structures to file layouts and addresses; and (c) translate logical data operations into low-level I/O operations for a variety of parallel file systems. Due to the dual translation steps, each task within the workflow can have unique data access patterns, and similar high-level data access patterns can translate into surprisingly divergent low-level I/O operations. Examples include unexpected layouts, metadata overhead, and data fragmentation. The result can be bottlenecks that are extremely difficult to diagnose.

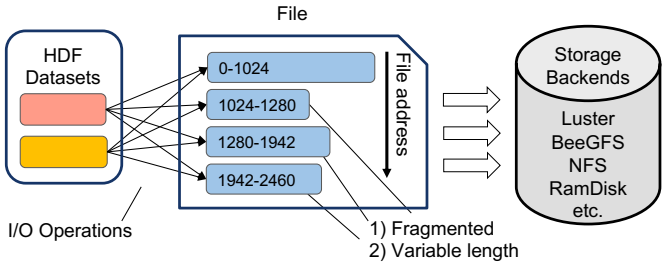


Figure 1: I/O problems identified in HDF for fragmentation and variable length.

Current I/O analysis techniques fail to offer insight into these challenges. System-level techniques — task scheduling, data caching, staging [7][8][9], and I/O system tuning — not only fail to capture the full potential of most applications but also fail to connect logical datasets to I/O. Application-specific profilers, such as Darshan[10] and Recorder[11], focus on analyzing individual I/O operations within individual applications, preventing (a) analysis of the datasets-to-I/O *connections* as well as (b) those relationships across the workflow's *multiple tasks*. The failure is especially acute when workflow tasks use different datasets and have varying I/O demands [12][13].

The key to efficient workflow data movement lies in gaining in-depth insights into data access patterns and I/O characteristics within the context of the entire workflow. To address this challenge, we introduce DaYu,[1] a data flow analysis and optimization framework. DaYu goes beyond traditional profilers by leveraging rich metadata from high-level I/O libraries like HDF5 [14] and netCDF [15]. This rich metadata enables DaYu to uncover (a) the semantic relationships between logical datasets and file mappings, (b) how dataset operations translate into I/O, and (c) the interactions across entire workflows. By analyzing application data usage and I/O behaviors associated with each logical dataset, DaYu can pinpoint potential bottlenecks and suggest strategies for improving data movement performance. Although we focus on HDF5, these challenges are common across other I/O libraries and descriptive data formats that utilize metadata and require translation from the abstract level to low-level I/O.

Our contributions can be summarized as follows:

[1]DaYu (e.g. "Yu the Great") refers to a legendary Chinese king credited with taming floods through water control projects.

2168-9253/24/$31.00 ©2024 IEEE
DOI 10.1109/CLUSTER59578.2024.00038

Authorized licensed use limited to: Illinois Institute of Technology. Downloaded on November 12,2024 at 02:59:36 UTC from IEEE Xplore. Restrictions apply.

- *Dynamic cross-application dataset insights* by capturing the mapping between a datum's logical and storage name and address for all I/O accesses, whether symbolic or not. Additionally, our method generates these mappings *across applications*, forming complete data dependence chains for all I/O accesses, whether targeting self-describing formats (e.g., HDF5 I/O) or block formats (e.g., POSIX I/O).
- *Workflow dataset bottleneck detection* using an interactive visualizer. This visualizer presents the mapping of task execution time and data volume within the context of the data flow dependence graph.
- The open-source DaYu workflow performance toolset. DaYu is lightweight and efficient, with an execution time overhead of less than 4% on time-sensitive data traces; and has small post-execution analysis requirements.
- *Performance evaluation* using benchmarks and scientific workflows with varying I/O patterns to assess DaYu's time and space overhead, as well as its ability to uncover previously unknown bottlenecks that arise from not fully appreciating the implications of DaYu's cross-application mapping.

II. Challenges of Descriptive Data Formats

Modern scientific workflows heavily rely on descriptive data formats, such as HDF5 [14], netCDF [15], and ADIOS [16], to provide enriched metadata, unified data structures, and high-performance I/O. However, as illustrated in Figure 1, these formats also create obscure I/O bottlenecks due to the need to translate high level operations into file addresses and then into low-level I/O operations. We describe several challenges with a focus on HDF5 as an prime example due to its popularity, active maintenance, advanced features, and extensive use across scientific domains [17]. The challenges we describe, however, are generalizable to other I/O libraries.

Challenge 1*: Obscured Low-Level I/O Behaviors within HDF5*. HDF5 manages data using a hierarchical model built upon data objects. A typical HDF5 file often has a group containing multiple datasets, with the option to attach attributes to datasets for additional context. Different data objects can exhibit different I/O behaviors. Many scientific I/O libraries exhibit similar behavior, where data layouts and resulting I/O patterns are determined by internal management and hidden from the user. The challenge illustrated in Figure 1 specifically highlights how the data content of two HDF5 datasets can be stored in many different file regions. Understanding these internal behaviors remains essential for performance optimization in various scientific workflows.

Challenge 2*: I/O Libraries are Agnostic about Application-Level Data Usage.* Application-level data usage can vary drastically across tasks within a workflow. However, I/O libraries are agnostic to these different data usage patterns, making them unable to determine the optimal storage layout for specific use cases. In particular, HDF5 offers two primary data storage layouts: contiguous and chunked, each with distinct I/O behaviors. Contiguous layouts are ideal when tasks use the entirety of a fixed-length dataset, as they allow for a single I/O operation to a continuous file region. In contrast, chunked layouts, which divide datasets into smaller storage chunks, offer flexibility for non-sequential I/O and opportunities for parallelization when tasks access random or partial regions of a dataset.

Challenge 3*: Fragmentation in Data Storage.* Fragmentation introduces additional and irregular data accesses, hurting performance. There are two primary causes of storage fragmentation in HDF5: 1) chunked layout; 2) variable-length (VL) data. In both cases, fragmentation stems from storing index data and the actual data in separate file regions. VL data, commonly seen in scientific sparse data, images, and text data, exacerbates fragmentation across file regions due to its inherent size variability. To address this, I/O libraries often employ special storage techniques, such as storing element lengths alongside the data or using delimiters to separate records. Additionally, techniques like padding VL data to a fixed size or structuring it for predictable access can improve I/O performance, but at the cost of increased data size.

III. Overview

DaYu is a framework designed to help application and workflow performance analysts understand data flow behavior. It extracts workflow data patterns and develops insights into the behavior of data flows, providing opportunities for optimizing application I/O. These insights also benefit application users and I/O library developers. Its source code is available on GitHub [18].

The DaYu framework comprises three primary components:

1) *Data Semantic Mapper*, which maps semantic datasets to I/O statistics, capturing essential data flow insights for analysis. (section IV).
2) *Workflow Analyzer*, which groups I/O statistics by high-level data semantics and visualizes the combination as semantic dataflow graphs, to provide insights into holistic data dependence for I/O accesses (section V).
3) *Data Flow Diagnostics*, which explores three real-world scientific workflows from distinct domains, generating visualizations of dataflow and I/O semantics, revealing potential I/O improvement opportunities, and which is empowered with optimizations suggested by DaYu's insights (section VI).

A. Guiding Datasets and Workflow I/O Optimization

Guided by DaYu's diagnostic insights, performance analysts can employ the following guidelines to optimize workflow I/O.

1) Customized Caching: To address **Challenges 1, 2, and 3**, DaYu unveils connections between high-level semantics and low-level I/O (e.g., POSIX), uncovering insights into data reuse. This suggests prioritizing frequently used data in the fastest available storage or in memory with data buffer middleware like Hermes [19] to reduce storage data accesses and latency. This applies to both intra- and inter-task data reuse.

Authorized licensed use limited to: Illinois Institute of Technology. Downloaded on November 12,2024 at 02:59:36 UTC from IEEE Xplore. Restrictions apply.

2) Partial File Access: To address **Challenges 1, 2, and 3**, DaYu enables identification of opportunities for partial data access within HDF5 files. This is particularly valuable for cases like VL data, where only a subset of the data is required. Using middleware I/O techniques like Hermes [19], we can reduce unnecessary data movement by only accessing specific file segments. Selective caching is crucial when resources are limited, as it saves bandwidth, memory usage, and I/O.

3) Customized Prefetching: To address **Challenge 1**, DaYu reveals low-level I/O behaviors across tasks. We can leverage that to develop customized prefetching for optimized data locality and latency with the following guidelines. Notably, the customized prefetching guidelines (see below) are generic and can be applied to any workflows.

- When DaYu anticipates a task's data requirements, data prefetching can be employed. This could involve placing the data on faster storage (SSDs) or in memory (Hermes [19]) for improved data access.
- When the network is congested, determining the optimal prefetching timing is important to reduce latency.
- When data is shared by multiple tasks across different compute nodes, staging it to node-local storage reduces data access latency. Additionally, this also reduces the concurrency per file, alleviating I/O contention.

4) HDF5 Data Format Optimization: To address **Challenges 2 and 3**, DaYu reveals the I/O behaviors associated with different data layouts. Using the following guidelines, we can determine the most efficient data layout.

- *Small, Fixed-Length Data*: To reduce the number of I/O operations by reading the entire dataset at once, select contiguous layout.
- *Large, Fixed-Length Data*: 1) Select contiguous layout to optimize for sequential access; 2) Select chunked layout to optimize for random or parallel access.
- *Variable-Length Data*: For any data size, Select chunked layout to leverage metadata for efficient random file I/O access.

IV. Connecting Datasets with I/O

To gain in-depth workflow data flow insights, the *Data Semantic Mapper* connects the high-level semantics of data interactions ("what") with their underlying I/O behaviors ("how"). Currently, there is a lack of profiling tools to interface with HDF5, collect both high-level data accesses and low-level I/O operations, and to map between them. Capturing and connecting both high-level and low-level details is crucial to painting a comprehensive picture of workflow data interactions between tasks and I/O performance. To bridge this gap, *we created the DaYu tool to demonstrate our methodology for connecting HDF5 and netCDF high-level data semantics with low-level I/O behaviors.*

Profiling: To bridge this gap, DaYu implemented a profiling tool that leverages HDF5 APIs to monitor data access patterns and I/O operations at runtime. DaYu implements a two-layer external HDF5 plugin through the existing APIs, namely the

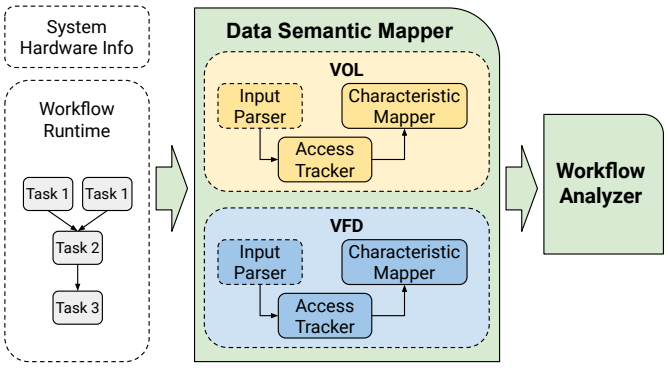


Figure 2: Data Semantic Mapper High-Level Design.

Virtual Object Layer (VOL) [20] and the Virtual File Driver (VFD) [21]. As illustrated in the high-level design of the Data Semantic Mapper (Figure 2), each of the two layers has three main components: (1) *Input Parser*; (2) *Access Tracker*; and (3) *Characteristic Mapper*.

The HDF5 plug-ins allow real-time monitoring of various time-sensitive profiles at high-level (VOL profiler) and low-level (VFD profiler), summerized in Table I and Table II respectively.

Input Parser: This component reads the user-provided configuration and parameters for initialization. For example, the location to store the recorded statistics, the page size to record, the number of I/O operations to skip, and whether to turn on/off I/O tracing. This flexibility allows users to adjust the data collection granularity, reducing storage overhead based on their analysis needs. The workflow launcher or application must inform DaYu of the current task.

Access Tracker: Data access information is collected at two levels.

The HDF5 data object level is referred to as high level, characterized by data captured through the VOL profiler. DaYu data object profiles measure key characteristics, including data type, shape, and access patterns. In particular, we record the data object name, associating it with the specific file and the current task.

HDF5 generates I/O in its lower layer, characterized by data captured through the VFD profiler. DaYu's I/O profiles measure key characteristics, including I/O operations, timing information, and file statistics. We record the I/O operations, associating them with the specific file and the current task.

All these statistics are collected as entries in a hash table in the duration of the task.

Characteristic (VOL-VFD) Mapper: Due to HDF5's internal management, the mapping between data access and I/O operations is obscured. This hides behaviors like a single data object access that triggers multiple, dispersed I/O operations due to factors such as chunked layout or variable-length data. Specifically, the VOL profiler is unaware of the logical file address accessible through the VFD profiler. To address this challenge, DaYu establishes a mapping between high-level data objects and their corresponding low-level I/O operations.

Authorized licensed use limited to: Illinois Institute of Technology. Downloaded on November 12,2024 at 02:59:36 UTC from IEEE Xplore. Restrictions apply.

Parameter	Description	Goal
1. Task Name	The current task name	Create file-task relationship
2. File Name	List of filenames interacted by the task	
3. Object Name	The data object acessed by task	Map I/O operations to Data Object
4. Object Lifetime	List of lifetimes: $T_{object_release} - T_{object_aquired}$.	Maintain temporal relationships
5. Object Description	Data object details (shape, type, size, etc)	Enrich data object semantics
6. Object Access	Data object read/write by the task	Record application memory/object utilization

Table I: VOL Profiler Object-Level Semantics

Parameter	Description	Goal
1. Task Name	The current task name	Create file-task relationship
2. File Name	The file name the task is accessing	Create file-task relationship
3. File Lifetime	Access time $T_{close} - T_{open}$ for the given file	Map I/O operations to the task
4. File Statistics	Traditional metrics (size, count, sequential, etc)	Capture access pattern to different regions
5. I/O Operations	The I/O operation count and the file address region	The low level (e.g. POSIX) I/O behavior
6. Access Type	Flag indicating metada vs data operations	Capture the file region acessed
7. Data Object	The data object name acessed by the task and file	Map I/O operations to Data Object

Table II: VFD Profiler File-Level Semantics

The *mapping* is achieved by establishing a connection between DaYu's VOL and VFD profilers. This connection associates data objects with their corresponding I/O operations. Unfortunately, due to HDF5's abstraction layer, achieving direct internal communication between the VOL and VFD layers is inherently difficult. We enable the mapping by utilizing shared memory between the two layers, allowing DaYu to communicate the current data object to the lower-level profiler when recording I/O. With this mapping, DaYu reveals the distinct I/O behaviors of individual data objects. These behaviors are then formulated as I/O statistics grouped by data objects/layouts.

We can further categorize I/O operations into metadata and raw data operations based on data access types (parameter 6 in Table II). This categorization reveals the distinct I/O behaviors exhibited by each type: 1) Metadata I/O often involve smaller file regions and I/O sizes; 2) Raw data I/O can exhibit more diverse behaviors.

V. Connecting Datasets Across Workflows

To optimize workflow performance, we must connect data accesses to workflow tasks in ways that expose I/O issues and reveal potential optimizations. Raw I/O traces generated by traditional profilers lack methods to incorporate workflow data movement, semantics, and dependencies, all of which are critical for revealing bottlenecks and insights. To bridge this gap, DaYu's *Workflow Analyzer* connects data-to-task into a workflow graph and decorates the graph with data semantics and I/O statistics.

To avoid overly complex graphs, the *Workflow Analyzer* generates two types of graphs: one for a complete overview, and another for deeper semantic analysis.

File-Task Graphs (FTGs): This graph depicts interactions between data files and tasks, with files and tasks represented as nodes, and directed edges signifying read or write operations. Connecting the task and file nodes reveals several key profiles of the workflow:

- Task and file dependencies;
- Task I/O operations, time ordered file access, and execution timing;
- I/O datum production and consumption across tasks.

FTGs are constructed based on the execution order of tasks. The parameters collected in Table II allow us to construct a task and file dependency graph, forming the foundation for an FTG. Notably, while current FTG construction requires manual input for task ordering, future DaYu versions will automate this process by integrating with workflow management tools. The edges between connected nodes then include the collected data access information. An example FTG of the PyFLEXTRKR workflow is shown in Figure 4.

Semantic Dataflow Graphs (SDGs): *This graph build upon FTGs by introducing a HDF5 data object layer between files and tasks. This provides insights into the semantic data within the workflow, revealing how data objects interact with files and tasks.* SDGs can be further enhanced with nodes representing logical file addresses/regions (an capability offered by the *Workflow Analyzer*) to show where a dataset's content distributes within the file. Connecting the file, data, and task nodes reveals additional key profiles of the workflow:

- Task and data object dependencies, access ordering;
- Data object usage across different tasks;
- Data object and file relationships;
- Data object mapping to file and to different file regions.

Finally, the SDG graph is enriched with data access statistics (access count, data volume, bandwidth), which differentiates between data and metadata accesses.

An example of *enhanced* SDG is shown in Figure 3. Figure 3 depicts a single-task workflow using blue file nodes and red task nodes. The data flow (from left to right) shows data-related nodes (datasets, addresses, files) appearing after task

Authorized licensed use limited to: Illinois Institute of Technology. Downloaded on November 12,2024 at 02:59:36 UTC from IEEE Xplore. Restrictions apply.

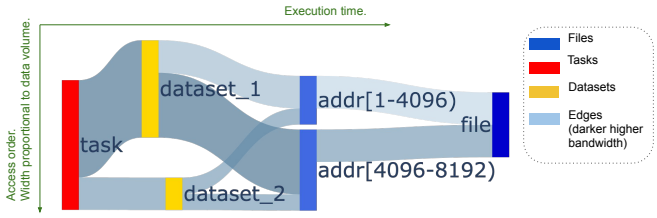


Figure 3: An example SDG. The nodes are arranged vertically by event start time (top to bottom) and horizontally by event end time (left to right). Dataset nodes (*dataset_1*, *dataset_2*) represent data objects that contains actual data, while *addr* nodes indicate specific file address regions.

nodes for write operations. Dataset nodes (yellow) precede the task node, and lighter blue nodes indicate file address regions within the darker blue file node. For clarity, the *Workflow Analyzer* divides addresses into regions based on a customized page size (e.g., $[0-4096)$ for 4096 pages; each is one byte). Edge color denotes I/O bandwidth (lighter is lower bandwidth), and node/edge width represents data volume. Additionally, both the FTG and SDG edges contain detailed data access statistics (interactable in the HTML format), allowing differentiation between HDF5 metadata and raw data accesses.

Adjusting Resolution: When SDGs become complex due to workflows with numerous tasks and parallel execution, the *Workflow Analyzer* enhances readability by presenting a less complex graph. It allows users to group and aggregate nodes by time, space, task, or location dimensions, either during the analysis step or within the interactive HTML format graph.

VI. Data Flow Diagnostic

This section analyzes the visualized I/O patterns of three scientific workflows: *PyFLEXTRKR*, *ARLDM*, and *DDMD*. Using real-world workflow graphs and profiles generated by DaYu *Workflow Analyzer*, we uncover unique insights and identify unknown bottlenecks.

A. Storm Tracking

1) Workflow Overview: The storm tracking scientific workflow involves two distinct phases. The first phase simulates the turbulent flow using specialized models like large eddy simulations (LES). The second phase focuses on data analysis using feature tracking software. For this study, we focus on the analysis phase, utilizing the advanced PyFLEXTRKR software [22]. PyFLEXTRKR offers diverse feature-tracking algorithms for various spatial and temporal scales of atmospheric sensor data, enabling in-depth analysis of storm development.

In a workflow, stages represent logical groupings of tasks designed to achieve distinct milestones within a larger process. PyFLEXTRKR's workflow comprises nine sequential stages, where tasks within each stage can be executed in parallel across distributed computing environments. A master workflow manager orchestrates task deployment, scheduling, and execution. Functionally, the PyFLEXTRKR workflow divides into two parts. Initial stages focus on feature identification and mapping, archiving results; latter stages emphasize analysis, requiring frequent, complex interactions with large datasets. PyFLEXTRKR's complex data dependencies and irregular, repetitive access patterns, common in feature-based image analysis and cluster-based anomaly detection, make it an ideal case study for exploring common practice scientific workflow I/O behavior.

2) Observations: Figure 4 shows the FTG of the nine-stage PyFLEXTRKR pipeline, illustrating the first three observations. Figure 5 provides a detailed SDG focused on stage-9, which exemplifies the fourth observation. In Figure 4, stages are ordered from left to right, visually representing the task execution sequence. Thus, *run_idfeature* is considered a stage-1 task, and *run_speed* a stage-9 task.

- Data reuse: The stage-3 task exhibits write-after-read data access. Stage-1 task output is used by multiple downstream tasks in stages 2, 3, 4, 6, and 8.
- Time-dependent inputs: Observe that the stage-6 task's input files are only required in the middle of the workflow, right before the task.
- Disposable data: Once processed, initial input files (for stage-1 task) and outputs from tasks with only one outgoing edge (marked in blue) become non-critical for further workflow steps.
- Data scattering: Figure 5 exposes an I/O bottleneck in stage-9: many small datasets (less than 500 bytes) within a file. This causes frequent metadata access, generating excessive small I/O requests.

3) Optimizations: For each observation above, we can employ the below optimizations respectively suggested by DaYu's optimization guidelines (Section III-A).

- Customized caching: We can cache the targeted data in the fastest access tier (e.g., memory or node local storage).
- Customized prefetching: Delaying data prefetch until before it is needed can reduce network congestion overhead, particularly when resource is limited.
- Data Stage-out: Offload files to slower storage, freeing space for data essential to later tasks.
- Data Format Optimization: For small and fixed-length data, we can reduce I/O operation by compacting them into one large dataset.

B. Molecular simulation with Deep Learning

1) Workflow Overview: DeepDriveMD (DDMD) is a scientific workflow for protein folding that integrates simulation, machine learning (ML) training, and inference in a single pipeline [23]. This hybrid approach reflects the expanding use of ML analysis across scientific domains, such as materials science [24] and climate modeling [25].

DDMD employs an iterative process with four distinct stages: simulation, aggregation, training, and inference. Each iteration leverages insights from the previous one to guide subsequent experiment configuration. A single DDMD iteration follows a 4-stage pipeline: OpenMM simulation (12 parallel tasks), aggregation, training, and inference (each with

Authorized licensed use limited to: Illinois Institute of Technology. Downloaded on November 12,2024 at 02:59:36 UTC from IEEE Xplore. Restrictions apply.

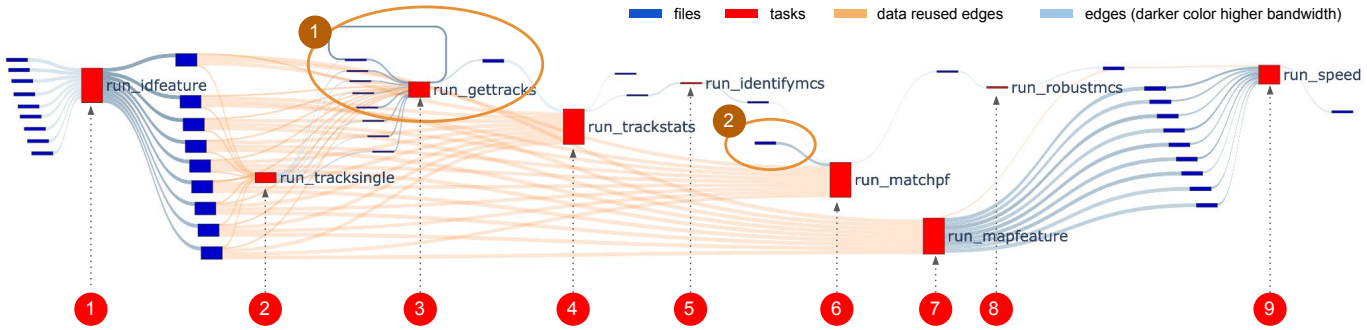


Figure 4: PyFLEXTRKR Workflow FTG. (1) Circle 1 highlights the write-after-read pattern of task *run_gettracks* (stage-3). (2) Circle 2 identifies input files that are not accessed at the start of the workflow. (3) Orange edges shows the data node with more than two out-going edge, which means data reuse. Red circles denote stage numbers.

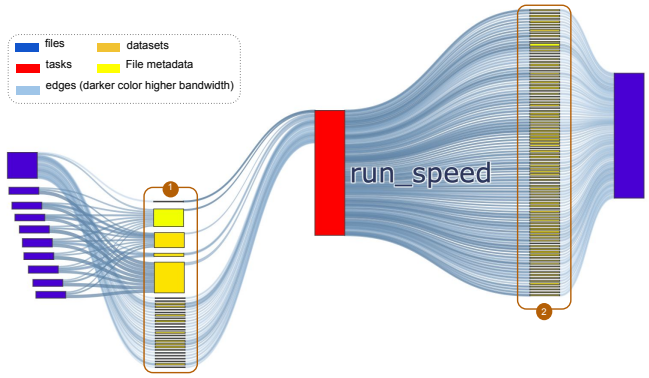


Figure 5: Pyflextrkr Stage-9 SDG. Both (1) and (2) shows many small datasets exists in each file, the number of edges here present the actual number of datasets in all files, which is not bounded and can become large.

1 task). In the OpenMM stage, each task generates HDF5 files containing four datasets (*contact_map*, *point_cloud*, *fnc*, *rmsd*). Simulation and ML analysis integration is becoming prevalent in scientific workflows [26], and DeepDriveMD exemplifies the I/O patterns of such workflows, which often exhibit diverse data access characteristics – a common feature in HPC simulation-analysis-visualization workflows [27], [28].

2) Observations: Figure 6 shows the FTG of the full DDMD workflow for the first 3 observations. Figure 7 shows the SDG of the aggregate and training stage for the 4th observation.

- Read-only access: The aggregate and inference stages read all simulated data, and accessing these files sequentially.
- Data reuse: The *training* task shows read-after-write pattern on specific embedding files (numbers 5 and 10).
- None-data dependent tasks: We see training and inference stages have no HDF5 data dependency as they show independent input and output files.
- Partial file access: From Figure 7 we observe the *aggregate* task consolidates four datasets and file metadata from simulated data into a single file, but doesn't modify the underlying data. Interestingly, the subsequent training stage only uses three of these datasets, excluding the largest dataset *contact_map*. This suggests that a significant portion of data processed during aggregation is ultimately unused in training.
- Metadata overhead: Although the graph doesn't show data semantics details, our analysis reveals that all datasets involved in the workflow are of HDF5 chunked layout. However, the chunked layout becomes inefficient for the workflow's small file sizes. Chunking HDF5 files into small fragments adds metadata overhead and incurs extra I/O operations.

3) Optimizations: Below are the optimizations suggested by DaYu's guidelines for each observations respectively.

- Customized prefetching: To improve data locality, prefetch the simulated data to the fastest storage tier close to the aggregation and inference tasks. Schedule these tasks on the same compute resource. Since the input files are accessed sequentially, implement a rolling stage-in strategy for efficient data movement.
- Customized caching: Cache the specific training task files to reduce access latency.
- Task Parallelization: We can parallelize training and inference tasks without data dependencies. In DDMD, inference relies on the training-generated model, a dependency DaYu's HDF5 layer focus doesn't fully capture. Despite the model dependency between iterations, there are optimization opportunities due to data independence within each iteration. A pre-trained model enables parallel inference and training in the current iteration, with the inference stage subsequently using the previous iteration's trained model.
- Partial File Access: We can selectively access the used datasets in the aggregate stage and leave out the unused dataset to reduce data movement.
- Data Format Optimization: Change the fragmented chunked dataset to contiguous data layout to reduce I/O operation.

Authorized licensed use limited to: Illinois Institute of Technology. Downloaded on November 12,2024 at 02:59:36 UTC from IEEE Xplore. Restrictions apply.

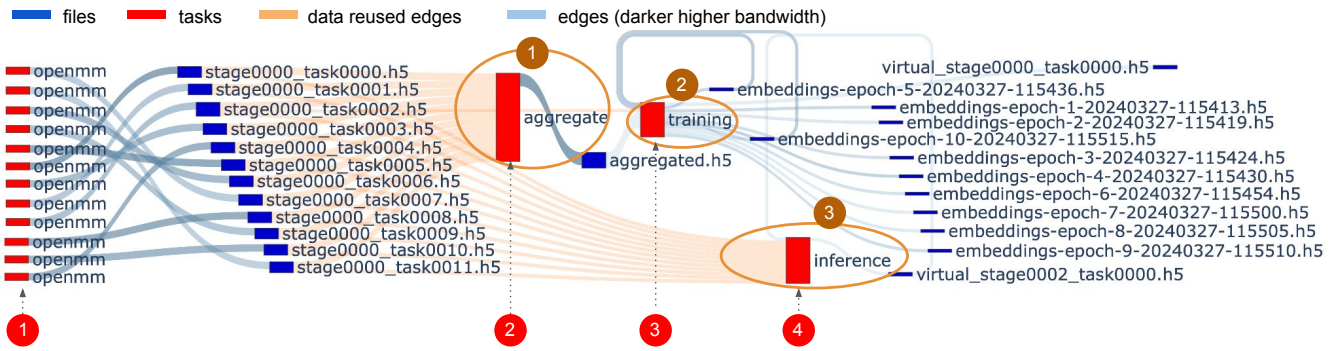


Figure 6: The DeepDriveMD workflow FTG depicts a 4-stage pipeline. Red circles denote stage numbers. (1) Both the *aggregate* task (Circle 1) and *inference* task (Circle 3) access all simulated data. (2) In contrast, the *training* task (Circle 2) primarily reads from the "aggregated" output and accesses only one simulated data file. (3) Notably, the graph reveals no direct data dependency between the *training* and *inference* tasks.

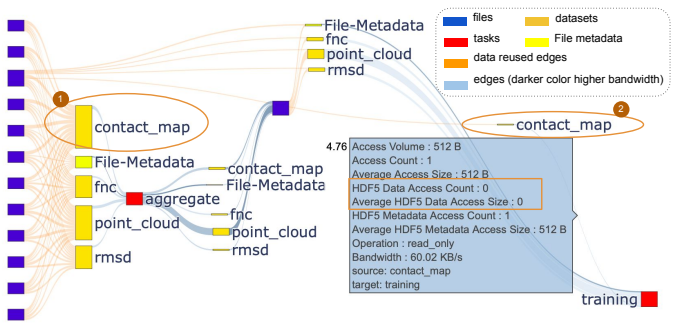
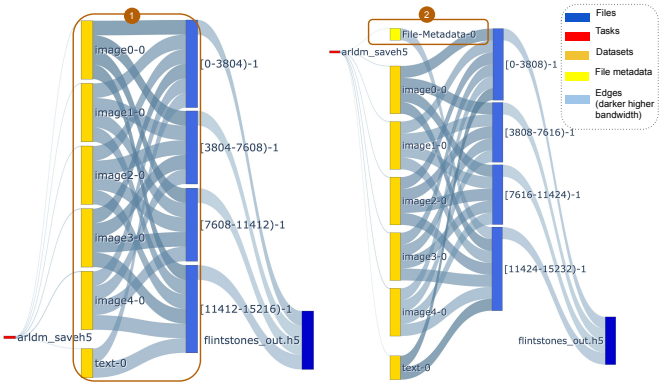


Figure 7: The DeepDriveMD partial workflow SDG shows dataset usage between the *aggregate* and *training* tasks. (1) Among the datasets, the *contact_map* (Circle 1) has the largest volume. (2) The *training* task accesses aggregated data but does not directly utilize the *contact_map*. Instead, as shown in Circle 2, the *contact_map* is read from simulation output. (3) The pop-up (highlighted in orange) indicates that the *training* task only accesses the *contact_map*'s metadata, not its data content.



(a) Contiguous Datasets SDG. (b) Chunked Datasets SDG.

Figure 8: Figures (a) and (b) depict the SDG of the stage-1 task *arldm_saveh5* within the ARLDM workflow. (1) Box 1 highlights the fragmentation of datasets across different logical file address regions. This fragmentation is consistent between both figures. (2) Box 2 in (b) highlights the file metadata region, a distinctive feature of chunked dataset layouts.

C. Image Synthesis

1) Workflow Overview: Advanced diffusion models like DALL·E and Stable Diffusion have revolutionized machine learning with their ability to generate images from text descriptions. Building upon these advancements, the Auto-Regressive Latent Diffusion Model (ARLDM) incorporates historical context and multimodal conditions for enhanced understanding and image synthesis capabilities [29]. The ARLDM workflow consists of three stages: First, the data preparation stage, where image and text data are prepared and stored in HDF5 format (with images in separate datasets). Next, the training stage, where the model reads image datasets from HDF5 for training. Finally, inference is performed on datasets.

ARLDM uses a 1D array of variable-length data for image and text storage. We chose this workload due to the prevalence of variable-length sequences in NLP models and the similar management and performance issues shared by sparse data, common in scientific programming [30].

2) Observation: Figure 3 depicts the first stage of the workflow, in which a data preparation task *arldm_saveh* on the leftmost node writes to a file *flinstone_out.h5* on the rightmost node. The later stages are omitted due to similar analysis. The yellow nodes are HDF5 datasets, while the lighter blue nodes represent file address regions, connecting to the darker blue file node. In ARLDM, file addresses are divided into four regions, each indexing a range of 64KB file pages. For example, the node $[0-3804)$ represents file addresses from page 0 to page 3804. The same interpretation applies to the remaining address ranges.

Lack of metadata: Figure 8a shows the data flow with the default data layout. In box 1, the task *arldm_saveh* writes to six HDF5 datasets, each mapped to specific file page ranges.

Authorized licensed use limited to: Illinois Institute of Technology. Downloaded on November 12,2024 at 02:59:36 UTC from IEEE Xplore. Restrictions apply.

Machine	Compute, Memory	Storage options (notes)
CPU cluster	2x Intel(R) Xeon Silver 4114, 48 GB RAM	NFS (default); NVMe SSD (node); SATA SSD (node); HDD (node)
GPU cluster	2× AMD EPYC; NVidia RTX 2080 Ti; 384 GB RAM	NFS (default); BeeGFS (w/ caching); SSD(node)

Table III: Machine configurations for experiments.

This spread implies that each dataset has content in all four different file regions. Notably, all datasets share the first region (the default location for metadata), while the actual VL data content of each dataset is stored in different file regions.

3) Data Format Optimization: Data flow with a chunked layout is illustrated in Figure 8b. Box 2 highlights HDF5's File-Metadata region, which stores metadata about all datasets within the file. External optimization systems can leverage this information for data placement and identifying potentially unused data. Interestingly, the chunked layout uses only slightly more file address space (chunked up to page 15232 vs. contiguous up to 15216) due to this additional metadata. Our I/O analysis reveals significantly fewer I/O operations when reading chunked datasets – specifically, half the number of POSIX write operations compared to contiguous layouts. This improvement results from the availability of metadata information when handling variable-length data.

VII. Evaluation

A. Experimental Setup

Hardware: Our evaluation uses the machines listed in Table III. While our evaluation focuses on medium-to-small scales, the findings remain relevant for large-scale workflows. Control loops often operate at these scales, and smaller deployments can present more performance optimization opportunities due to less data movement and resource contention [31].

Evaluation Software: We employ various evaluation methods to validate DaYu's effectiveness. Our DaYu tool gathers and analyzes runtime data access patterns to identify areas for optimization. Internally, DaYu relies on HDF5 for tracking application data access. For a comprehensive overhead assessment, we use the well-established H5bench benchmark suite [32], a representative parallel I/O benchmark designed for large-scale HDF5 workflows. We supplement H5bench with a custom Python script for simulating corner-case I/O behaviors.

B. Data Semantic Mapper Overhead Analysis

To evaluate the overhead introduced by DaYu's profiling capabilities, we conducted two tests: one with h5bench and one with a custom Python benchmark. Both tests were executed on a single node to maximize visibility of potential overhead. They capture, respectively, the typical and worst-case overhead incurred by DaYu's full functionality without data collection granularity adjustments (results in Figure 9).

The Python benchmark creates a corner-case scenario with an unusually large number (200) of datasets [33] stored in a small file. This test aims to isolate potential bottlenecks within DaYu's profiling, particularly when dealing with frequent data object access within a single task. Repeated reads of the same datasets within the same task trigger increased overhead because DaYu tracks semantic data even for closed datasets, deferring logging until the file is closed.

Results in Figure 9a and 9b show that the Data Semantic Mapper's overhead remains below 0.23% and decreases with increasing file size or number of processes. This is because DaYu's overhead is proportional to data object operations (create/open/close) and application access patterns. When data objects are opened or closed once per file and application access involves large I/O sizes, DaYu's overhead remains low even as program I/O time increases.

Note that DaYu's time-sensitive I/O tracing comes with a trade-off. Runtime overhead increases with higher I/O activity (read/write) within a file's open/close period, potentially reaching up to 4% (Figure 9c). The storage overhead of VOL remains low at 0.2%, while VFD increases linearly with I/O operations (Figure 9d).

Importantly, for analyses that do not require time-sensitive I/O traces, users can turn off I/O tracing. This will result in constant storage overhead.

This suggests that DaYu's storage footprint remains acceptable for the valuable insights it provides. Even with the largest tested storage footprint, the offline components require minimal processing time for the collected statistics. The *Workflow Analyzer* takes less than 15 seconds to analyze a graph with 1k nodes and 6k edges, and less than 2 seconds to construct the corresponding FTG and SDG in HTML format.

1) Component Analysis: To gain further insight into the performance of DaYu's Data Semantic Mapper, we analyzed the time breakdown of its three components: the Input Parser (which reads the configuration), the Access Tracker (which intercepts data access and I/O), and the Characteristics Mapper (which maps data to I/O). We compared its performance under two scenarios: an h5bench test using a large file size (80GB) and 64 processes, and the previously described corner-case Python test. The results of this analysis are presented in Figures 10a and 10b.

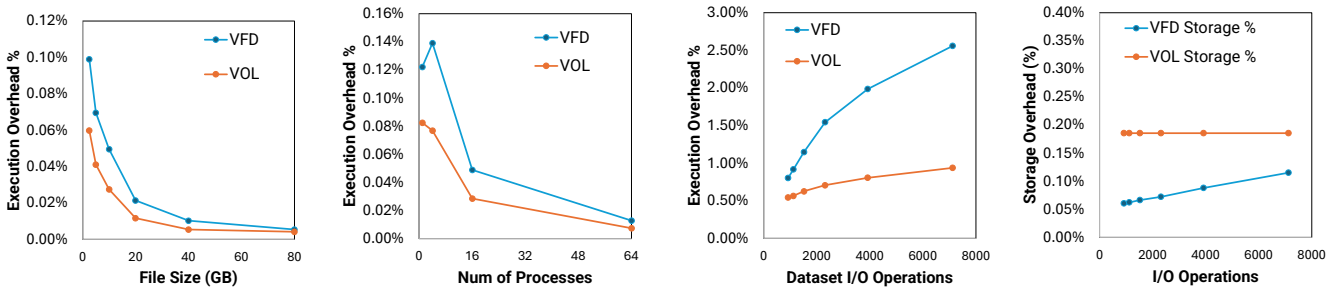
H5bench revealed minimal DaYu overhead, with 38.83 ms accounting for 0.008% of the execution time. The execution time primarily comes from the *Characteristics Mapper*. The corner-case Python test explored the worst-case scenario, showing 4% overhead (2.97% VFD, 1.0% VOL). This aligns with our observation that frequent data object open and close operations lead to higher overhead, particularly in the *Access Tracker* component, which records all data object accesses.

C. DaYu I/O Optimization Evaluation

1) Improving Data Placement: This evaluation demonstrates how DaYu's analysis can guide improvements in data locality, showcasing the potential for tailored data prefetching and task scheduling to optimize I/O according to established optimization guidelines.

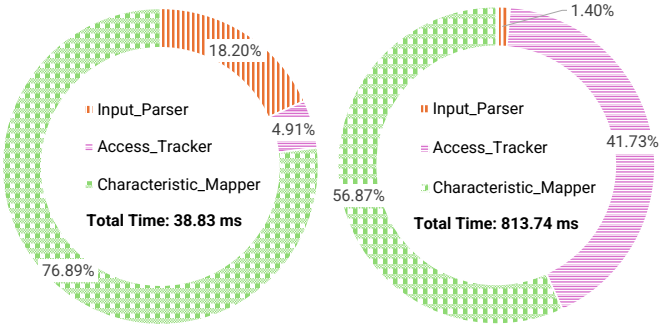
While large-scale evaluations are valuable, this work focuses on real-world applications that require faster turnaround times, often found at smaller scales. Here, data movement efficiency

Authorized licensed use limited to: Illinois Institute of Technology. Downloaded on November 12,2024 at 02:59:36 UTC from IEEE Xplore. Restrictions apply.



(a) H5bench data size scaling. (b) H5bench process scaling. (c) Python file read count scaling. (d) Python storage scaling.

Figure 9: DaYu's Data Semantic Mapper Overhead with VFD (low-level) and VOL (high-level): (a) H5bench with increased total file size; (b) H5bench with increased parallel I/O processes with a fixed 1GB I/O per process; (c) Increased dataset I/O count with a fixed 200MB file size; (d) DaYu statistics storage overhead compared to the program's required storage size.



(a) DaYu with h5bench. (b) DaYu with corner-case Python.

Figure 10: Breakdown of DaYu execution overhead.

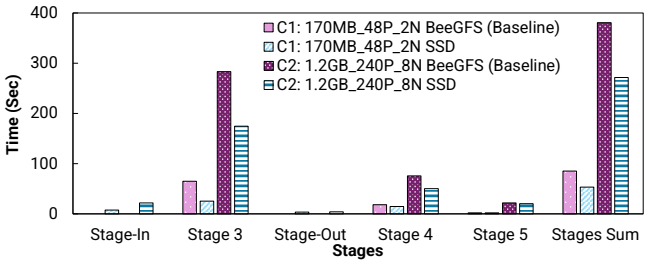


Figure 11: Comparing execution times for baseline vs. DaYu-optimized PyFLEXTRKR Stage 3-5 with Configuration 1 (C1) and Configuration 2 (C2) experiments. C1 uses a total of 170 MB of input files, 48 processes, and runs on 2 nodes. C2 uses a total of 1.2 GB of input files, 240 processes, and runs on 8 nodes.

limits optimization opportunities. Future work will explore how I/O performance scales with workflow size.

PyFLEXTRKR: DaYu's analysis shows that some files are heavily reused across stages, while others are specific to individual tasks within a stage. These insights enable strategic file placement to improve data locality. To demonstrate the impact, we evaluate the performance improvement on a sub-workflow encompassing stages 3 through 5 from the PyFLEXTRKR workflow (Figure 4). Focusing on stages 3-5, DaYu's analysis identifies that the *run_gettracks* tasks (stage 3) are parallelizable with an all-to-all access pattern (all tasks access all files). This function produces a single output file used as input for the subsequent *run_trackstats* task (stage 4). Stage 4 utilizes the same input files as stage 3, but the task is not parallelizable and exhibits a fan-in access pattern. By leveraging this knowledge, we can co-schedule stage 4 to run on the same node where the output from stage 3 is generated. Similarly, the *run_identifymcs* task (stage 5) is not parallelizable and accesses the output file from stage 4. DaYu identifies this one-to-one access pattern, enabling the co-scheduling of stages 3, 4, and 5 on the same processing node.

Our analysis identified two additional potential optimizations, but we opted not to implement them due to their negligible impact on overall performance. In the data reuse scenario (Circle 1, Figure 4, write-after-read I/O), the small file size allows it to be held entirely in memory during the task, ensuring it won't be swapped out. Since the workflow's current memory usage is not a constraint, implementing custom caching is unnecessary. Similarly, the time-dependent input files are also small and can be accommodated within the node's local storage. Consequently, prefetching data onto the node is not necessary at this stage to save space for them.

Our findings in Figure 11 demonstrate that, with a combination of data prefetching and task scheduling, the workflow runtime from stages 3 to 5 shows an overall speedup of 1.6x. Specifically, *Stage 3* in experiment C1 shows a speedup of *2.6x*.

DDMD: DaYu analysis revealed inefficiencies in the workflow. A large dataset created during *Aggregate* remains unused by *Training*, and both *Aggregate* and *Inference* share input files. To improve data movement, we can implement several optimizations following I/O optimization guidelines:

- Eliminate Unused Data Access: *Aggregate* will no longer access the unnecessary dataset, reducing data movement.
- Co-locate *Aggregate* and *Inference*: By placing *Aggregate* and *Inference* on the same node, I/O performance is

Authorized licensed use limited to: Illinois Institute of Technology. Downloaded on November 12,2024 at 02:59:36 UTC from IEEE Xplore. Restrictions apply.

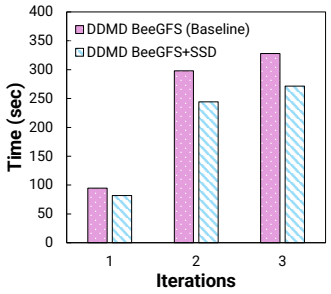


Figure 12: DDMD (12 tasks) execution vs. DaYu optimized.

improved, and data movement is significantly reduced.

- Pipeline *Training* and *Inference*: Training runs on a separate node with pre-staged input data, while *Inference* leverages the pre-trained model from the current iteration, optimizing execution flow.
- Asynchronous Data Staging: Finished data is asynchronously staged from local storage to shared storage during the startup of the next iteration, maximizing efficiency.

For the observed data reuse scenario (Circle 2, Figure 6, read-after-write I/O), the current experiment I/O sizes can be accommodated in memory during the task period without being swapped out by other I/O. Thus, implementing custom caching is unnecessary here. This approach yields a 1.15x performance improvement per pipeline iteration and a 1.2x improvement across a 5-iteration pipeline, as shown in Figure 12.

2) Improving Data Layout: This evaluation explores how DaYu's analysis guides the development of a straightforward data layout policy, ultimately reducing I/O times within workflows by following optimization guidelines.

PyFLEXTRKR: Our analysis of the workflow revealed a key inefficiency: scattered datasets were causing excessive metadata accesses. To address the challenge of metadata overhead, we implemented an optimization strategy that consolidates these small datasets into a single, larger one. This approach reduces metadata overhead by keeping track of the original file offsets within the consolidated dataset. To evaluate the I/O impact of data layout improvements, we simulated the stage-9 PyFLEXTRKR workflow (Figure 5) using various dataset sizes within a file. This file contained 32 datasets, each accessed 23 times. We tested performance under different numbers of concurrent processes to understand scaling behavior. The I/O was performed against a fast node-local NVMe drive to optimally handle small, concurrent I/O [34]. The measured I/O times (sum of POSIX operations) demonstrate that data consolidation significantly improves performance when datasets are small and numerous (Figure 13a). This optimization is particularly effective for files containing smaller datasets. Furthermore, I/O performance benefits increase as the number of processes concurrently accessing the same file decreases. Benchmarks across dataset sizes (1KB-8KB) and process scaling reveal I/O time reductions of 1.7x to 3.7x with data consolidation.

DDMD: Our analysis shows that the original chunked data layout causes excessive metadata and I/O operations. To improve I/O efficiency, we applied an optimization technique that converts datasets to a contiguous layout, reducing both metadata overhead and I/O operations. To measure the I/O time of these changes, we simulated the I/O behavior of DDMD's OpenMM and Aggregate tasks to evaluate the performance difference between chunked and contiguous dataset layouts.

As shown in Figure 13b, contiguous datasets consistently outperform chunked layouts across various dataset sizes and process counts. In high-concurrency scenarios typical of OpenMM, the performance improvement using contiguous datasets can achieve a speedup of 1.9x.

ARLDM: DaYu's analysis revealed that the ARLDM workflow frequently handles large datasets (6GB to 20GB), with over 90% being variable-length data. This data type requires extensive metadata to record data locations. To address this challenge and improve data access, we modified the default contiguous layout to HDF5's chunked layout. We measured the execution time of the task *arldm_saveh5*, which corresponds to the write time of the complete file.

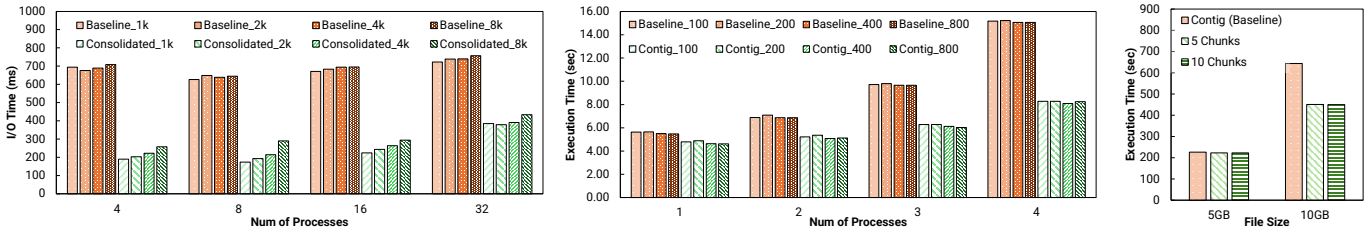
While contiguous and chunked layouts offer comparable performance for smaller datasets (as seen in the 5GB test case, Figure 13c), the advantages of chunking become significant with larger datasets. Our findings demonstrate that optimal chunk sizes can enhance write performance by up to 1.4x compared to contiguous layouts. This improvement stems from DaYu's analysis, which reveals that chunked layouts significantly reduce I/O operations (by 2x) for variable-length data compared to contiguous layouts. This reduction is due to the metadata within chunked datasets, which provides indexing for variable-length data and leads to better I/O performance.

VIII. Related Work

A. Application I/O Characterization

Significant effort has been devoted to characterizing application I/O behavior. Tools like Darshan [10] and Recorder [11] provide insights into storage-level or call-level I/O performance. However, they primarily focus on storage access characteristics rather than application-level data patterns. Additionally, tools like TOKIO [35] analyze extensive data traces for system-level insights but lack application-specific I/O analysis. While research has explored I/O patterns on supercomputers [36] and tools like IOMiner [37] identify root causes of poor I/O performance, they do not address cross-workflow data locality and reuse patterns. Existing tools and research highlight the value of capturing runtime I/O behavior, but a semantic-level approach is crucial for analyzing workflow-wide data access patterns and effectively communicating bottlenecks to programmers. This requires understanding data locality (reuse frequency) and reuse patterns (how data is utilized across tasks). Such analysis goes beyond traditional I/O profiling, uncovering insights into workflow I/O behavior

Authorized licensed use limited to: Illinois Institute of Technology. Downloaded on November 12,2024 at 02:59:36 UTC from IEEE Xplore. Restrictions apply.



(a) PyFLEXTRKR Baseline (Scattered) vs. Consolidated. (b) DDMD Baseline (Chunked) vs. Contiguous. (c) ARLDM.

Figure 13: Comparing the I/O performance of the original workflow data layout vs. the DaYu data layout.

that might otherwise be overlooked and pinpointing critical bottlenecks.

B. Workflow Characterization and Optimization

Characterization: Traditionally, scientific workflow characterization has focused on computational resource utilization, using resources like the Parallel Workload Archive [38] and the Grid Workload Archive [39]. While tools like Pegasus [40] capture task-to-file dependencies, they lack the data-centric information needed for improving I/O. Previous efforts by Bharath et al. and Juve et al. [41], [42] explored profiling workflows, including I/O and memory access characteristics. DaYu builds upon these works but offers a distinct approach by focusing on the relationship between tasks and data objects, enabling a new understanding of dataflow semantics within workflows.

Optimization: There is extensive research on optimizing application I/O, including tools like DXT Explorer [43] for log analysis that facilitates I/O optimization, Behzad et al.'s autotuning system [44], and Carretero et al.'s solution for reducing I/O contention with performance-aware I/O scheduling [45]. However, these primarily focus on optimizing single applications, leaving a gap in addressing cross-application I/O optimization. Lee et al. [46] introduced Data Flow Lifecycles to analyze workflows from task and data flow perspectives and proposed workflow optimization with this analysis. However, this approach lacks location and temporal ordering (per task and per file) as well as the semantic information needed to fully understand dataflow across applications.

IX. Conclusion

Currently, automated methods for understanding data flow between distributed tasks in scientific workflows are lacking, hindering effective I/O optimization. To address this challenge, we introduce DaYu, a data flow analysis and optimization framework. DaYu provides critical insights into data access patterns and I/O performance through a multi-layered data semantic extraction approach. The DaYu *Workflow Analyzer* extracts data flow and incorporates I/O behaviors to construct and visualize a Semantic Dataflow Graph. This comprehensive analysis enables users to uncover workflow optimizations. Guided by DaYu's insights, performance analysts can optimize workflows using the provided guidelines. Our evaluations demonstrate that DaYu's analysis enables significant I/O performance improvements, resulting in up to a 3.7x speedup. The time overhead for DaYu's time-ordered data is typically under 0.2% of runtime (4% in extreme cases). Storage overhead is under 0.25% of data volume (0.3% in extreme cases).

Enhancing DaYu's real-world impact is a key focus for future work. We plan to expand the I/O optimization guidelines, further leveraging DaYu's insights to automate optimization strategies. Our future work will involve integrating DaYu's analysis capabilities with existing I/O optimization ecosystems. This could include tools like Drishti [47] for performance analysis and optimization recommendations. Additionally, integration with I/O middleware like Hermes [19] could enable transparent and immediate runtime optimization, further streamlining the workflow optimization process. Future work will also investigate extending DaYu's support for asynchronous I/O and Message Passing Interface (MPI) applications to explore even broader applicability. Asynchronous I/O and MPI are prevalent in high-performance computing workflows, and enabling DaYu to analyze and optimize these workflows would significantly increase its impact.

X. Acknowledgments

This research is supported by the U.S. Department of Energy (DOE) through the Office of Advanced Scientific Computing Research's "Orchestration for Distributed & Data-Intensive Scientific Exploration"; the "Cloud, HPC, and Edge for Science and Security" LDRD at Pacific Northwest National Laboratory; and partly by the National Science Foundation under Grants no. CSSI-2104013 and OAC-2313154.

References

[1] G. Berriman, J. Good, A. Laity, A. Bergou, J. Jacob, D. Katz, E. Deelman, C. Kesselman, G. Singh, M.-H. Su *et al.*, "Montage: A grid enabled image mosaic service for the national virtual observatory," in *Astronomical Data Analysis Software and Systems (ADASS) XIII*, vol. 314, 2004, p. 593.

[2] L. Clarke, X. Zheng-Bradley, R. Smith, E. Kulesha, C. Xiao, I. Toneva, B. Vaughan, D. Preuss, R. Leinonen, M. Shumway *et al.*, "The 1000 genomes project: data management and community access," *Nature methods*, vol. 9, no. 5, pp. 459–462, 2012.

[3] E. Deelman, T. Peterka, I. Altintas, C. D. Carothers, K. K. Van Dam, K. Moreland, M. Parashar, L. Ramakrishnan, M. Taufer, and J. Vetter, "The future of scientific workflows," *The International Journal of High Performance Computing Applications*, vol. 32, no. 1, pp. 159–175, 2018.

Authorized licensed use limited to: Illinois Institute of Technology. Downloaded on November 12,2024 at 02:59:36 UTC from IEEE Xplore. Restrictions apply.

[4] J. Dongarra, P. Beckman, T. Moore, P. Aerts, G. Aloisio, J.-C. Andre, D. Barkai, J.-Y. Berthou, T. Boku, B. Braunschweig *et al.*, "The international exascale software project roadmap," *The international journal of high performance computing applications*, vol. 25, no. 1, pp. 3–60, 2011.

[5] O. Yildiz, M. Dorier, S. Ibrahim, R. Ross, and G. Antoniu, "On the root causes of cross-application i/o interference in hpc storage systems," in *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2016, pp. 750–759.

[6] H. Group *et al.*, "Hierarchical data format version 5," 1997.

[7] Q. Sun, T. Jin, M. Romanus, H. Bui, F. Zhang, H. Yu, H. Kolla, S. Klasky, J. Chen, and M. Parashar, "Adaptive data placement for staging-based coupled scientific workflows," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2015, pp. 1–12.

[8] P. Subedi, P. Davis, S. Duan, S. Klasky, H. Kolla, and M. Parashar, "Stacker: an autonomic data movement engine for extreme-scale data staging-based in-situ workflows," in *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2018, pp. 920–930.

[9] T. Wang, S. Byna, B. Dong, and H. Tang, "Univistor: Integrated hierarchical and distributed storage for hpc," in *2018 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 2018, pp. 134–144.

[10] P. Carns, K. Harms, W. Allcock, C. Bacon, S. Lang, R. Latham, and R. Ross, "Understanding and improving computational science storage access through continuous characterization," *ACM Transactions on Storage (TOS)*, vol. 7, no. 3, pp. 1–26, 2011.

[11] C. Wang, J. Sun, M. Snir, K. Mohror, and E. Gonsiorowski, "Recorder 2.0: Efficient parallel i/o tracing and analysis," in *2020 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 2020, pp. 1–8.

[12] H. Devarajan and K. Mohror, "Extracting and characterizing i/o behavior of hpc workloads," in *2022 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 2022, pp. 243–255.

[13] M. Liu, J. Zhai, Y. Zhai, X. Ma, and W. Chen, "One optimized i/o configuration per hpc application: leveraging the configurability of cloud," in *Proceedings of the Second Asia-Pacific Workshop on Systems*, 2011, pp. 1–5.

[14] M. Folk, G. Heber, Q. Koziol, E. Pourmal, and D. Robinson, "An overview of the hdf5 technology suite and its applications," in *Proceedings of the EDBT/ICDT 2011 workshop on array databases*, 2011, pp. 36–47.

[15] R. Rew and G. Davis, "Netcdf: an interface for scientific data access," *IEEE computer graphics and applications*, vol. 10, no. 4, pp. 76–82, 1990.

[16] W. F. Godoy, N. Podhorszki, R. Wang, C. Atkins, G. Eisenhauer, J. Gu, P. Davis, J. Choi, K. Germaschewski, K. Huck *et al.*, "Adios 2: The adaptable input output system. a framework for high-performance data management," *SoftwareX*, vol. 12, p. 100561, 2020.

[17] A. Collette, *Python and HDF5: unlocking scientific data*. " O'Reilly Media, Inc.", 2013.

[18] 2024. [Online]. Available: https://github.com/pnnl/DaYu

[19] A. Kougkas, H. Devarajan, and X.-H. Sun, "Hermes: A heterogeneous-aware multi-tiered distributed i/o buffering system," in *Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing*, ser. HPDC '18. New York, NY, USA: Association for Computing Machinery, 2018, pp. 219–230. [Online]. Available: https://doi.org/10.1145/3208040.3208059

[20] 2015. [Online]. Available: https://docs.hdfgroup.org/hdf5/develop/_h5_v_l__u_g.html

[21] J. Henderson and D. Robinson, *HDF5 VFD Plugins*, 2021. [Online]. Available: https://www.hdfgroup.org/wp-content/uploads/2021/10/HDF5-VFD-Plugins-HUG.pdf

[22] Z. Feng, J. Hardin, H. C. Barnes, J. Li, L. R. Leung, A. Varble, and Z. Zhang, "Pyflextrkr: a flexible feature tracking python software for convective cloud analysis," *EGUsphere*, pp. 1–29, 2022.

[23] H. Lee, M. Turilli, S. Jha, D. Bhowmik, H. Ma, and A. Ramanathan, "Deepdrivemd: Deep-learning driven adaptive molecular simulations for protein folding," in *2019 IEEE/ACM Third Workshop on Deep Learning on Supercomputers (DLS)*, 2019, pp. 12–19.

[24] J. Wei, X. Chu, X.-Y. Sun, K. Xu, H.-X. Deng, J. Chen, Z. Wei, and M. Lei, "Machine learning in materials science," *InfoMat*, vol. 1, no. 3, pp. 338–358, 2019.

[25] P. A. O'Gorman and J. G. Dwyer, "Using machine learning to parameterize moist convection: Potential for modeling of climate, climate change, and extreme events," *Journal of Advances in Modeling Earth Systems*, vol. 10, no. 10, pp. 2548–2563, 2018.

[26] A. Nouri, P. E. Davis, P. Subedi, and M. Parashar, "Exploring the role of machine learning in scientific workflows: Opportunities and challenges," *arXiv preprint arXiv:2110.13999*, 2021.

[27] C. Peña-Monferrer, R. Manson-Sawko, and V. Elisseev, "Hpc-cloud native framework for concurrent simulation, analysis and visualization of cfd workflows," *Future Generation Computer Systems*, vol. 123, pp. 14–23, 2021.

[28] J. Senk, A. Yegenoglu, O. Amblet, Y. Brukau, A. Davison, D. R. Lester, A. Lührs, P. Quaglio, V. Rostami, A. Rowley *et al.*, "A collaborative simulation-analysis workflow for computational neuroscience using hpc," in *High-Performance Scientific Computing: First JARA-HPC Symposium, JHPCS 2016, Aachen, Germany, October 4–5, 2016, Revised Selected Papers 1*. Springer, 2017, pp. 243–256.

[29] X. Pan, P. Qin, Y. Li, H. Xue, and W. Chen, "Synthesizing coherent story with auto-regressive latent diffusion models," *arXiv preprint arXiv:2211.10950*, 2022.

[30] J. Mainzer, N. Fortner, G. Heber, E. Pourmal, Q. Koziol, S. Byna, and M. Paterno, "Sparse data management in hdf5," in *2019 IEEE/ACM 1st Annual Workshop on Large-scale Experiment-in-the-Loop Computing (XLOOP)*, 2019, pp. 20–25.

[31] K. Mehta, A. Cliff, F. Suter, A. M. Walker, M. Wolf, D. Jacobson, and S. Klasky, "Running ensemble workflows at extreme scale: Lessons learned and path forward," in *2022 IEEE 18th International Conference on e-Science (e-Science)*, 2022, pp. 284–294.

[32] T. Li, S. Byna, Q. Koziol, H. Tang, J. L. Bez, and Q. Kang, "h5bench: HDF5 I/O Kernel Suite for Exercising HPC I/O Patterns," in *Proceedings of Cray User Group Meeting, CUG 2021*, 2021.

[33] T. H. Group, "Hdf5 dataset size and number questions," Apr 2024. [Online]. Available: https://forum.hdfgroup.org/t/hdf5-dataset-size-and-number-questions/12215

[34] Q. Xu, H. Siyamwala, M. Ghosh, T. Suri, M. Awasthi, Z. Guz, A. Shayesteh, and V. Balakrishnan, "Performance analysis of nvme ssds and their implication on real world databases," 2015.

[35] G. K. Lockwood, N. J. Wright, S. Snyder, P. Carns, G. Brown, and K. Harms, "Tokio on clusterstor: connecting standard tools to enable holistic i/o performance analysis," Lawrence Berkeley National Lab.(LBNL), Berkeley, CA (United States), Tech. Rep., 2018.

[36] T. Patel and S. Byna, "Uncovering access, reuse, and sharing characteristics of i/o-intensive files on large-scale production hpc systems." in *Proceedings of the 18th USENIX Conference on File and Storage Technologies, 2020*, 2020.

[37] T. Wang, S. Snyder, G. Lockwood, P. Carns, N. Wright, and S. Byna, "Iominer: Large-scale analytics framework for gaining knowledge from i/o logs," in *2018 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 2018, pp. 466–476.

[38] 2014. [Online]. Available: https://www.cs.huji.ac.il/labs/parallel/workload/

[39] 2024. [Online]. Available: http://gwa.ewi.tudelft.nl/

[40] E. Deelman, K. Vahi, G. Juve, M. Rynge, S. Callaghan, P. J. Maechling, R. Mayani, W. Chen, R. F. Da Silva, M. Livny *et al.*, "Pegasus, a workflow management system for science automation," *Future Generation Computer Systems*, vol. 46, pp. 17–35, 2015.

[41] S. Bharathi, A. Chervenak, E. Deelman, G. Mehta, M.-H. Su, and K. Vahi, "Characterization of scientific workflows," in *2008 third workshop on workflows in support of large-scale science*. IEEE, 2008, pp. 1–10.

[42] G. Juve, A. Chervenak, E. Deelman, S. Bharathi, G. Mehta, and K. Vahi, "Characterizing and profiling scientific workflows," *Future generation computer systems*, vol. 29, no. 3, pp. 682–692, 2013.

[43] J. Ousterhout and F. Douglis, "Beating the i/o bottleneck: A case for log-structured file systems," *ACM SIGOPS Operating Systems Review*, vol. 23, no. 1, pp. 11–28, 1989.

[44] B. Behzad, S. Byna, Prabhat, and M. Snir, "Optimizing i/o performance of hpc applications with autotuning," *ACM Transactions on Parallel Computing (TOPC)*, vol. 5, no. 4, pp. 1–27, 2019.

[45] J. Carretero, E. Jeannot, G. Pallez, D. E. Singh, and N. Vidal, "Mapping and scheduling hpc applications for optimizing i/o," in *Proceedings of the 34th ACM International Conference on Supercomputing*, 2020, pp. 1–12.

Authorized licensed use limited to: Illinois Institute of Technology. Downloaded on November 12,2024 at 02:59:36 UTC from IEEE Xplore. Restrictions apply.

[46] H. Lee, L. Guo, M. Tang, J. Firoz, N. Tallent, A. Kougkas, and X.-H. Sun, "Data flow lifecycles for optimizing workflow coordination," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2023, pp. 1–15.

[47] J. L. Bez, L. B. N. Laboratory, H. Ather, L. B. N. Laboratory, S. Byna, and L. B. N. Laboratory, "Drishti: Guiding End-Users in the I/O Optimization Journey," 2022.

Authorized licensed use limited to: Illinois Institute of Technology. Downloaded on November 12,2024 at 02:59:36 UTC from IEEE Xplore. Restrictions apply.