CVoR Issued, January 23, 2024. DOI: 10.1145/3581784.3607104

Data Flow Lifecycles for Optimizing Workflow Coordination

Hyungro Lee
Pacific Northwest National Laboratory
Richland, Washington, USA
hyungro.lee@pnnl.gov

Luanzheng Guo
Pacific Northwest National Laboratory
Richland, Washington, USA
lenny.guo@pnnl.gov

Meng Tang
Illinois Institute of Technology
Chicago, Illinois, USA
mtang11@hawk.iit.edu

Jesun Firoz
Pacific Northwest National Laboratory
Seattle, Washington, USA
jesun.firoz@pnnl.gov

Nathan R. Tallent
Pacific Northwest National Laboratory
Richland, Washington, USA
tallent@pnnl.gov

Anthony Kougkas
Illinois Institute of Technology
Chicago, Illinois, USA
akougkas@iit.edu

Xian-He Sun
Illinois Institute of Technology
Chicago, Illinois, USA
sun@iit.edu

ABSTRACT

A critical performance challenge in distributed scientific workflows is coordinating tasks and data flows on distributed resources. To guide these decisions, this paper introduces *data flow lifecycle* analysis. Workflows are commonly represented using directed acyclic graphs (DAGs). Data flow lifecycles (DFL) enrich task DAGs with data objects and properties that describe data flow and how tasks interact with that flow. Lifecycles enable analysis from several important perspectives: *task*, *data*, and *data flow*. We describe representation, measurement, analysis, visualization, and opportunity identification for DFLs. Our measurement is both distributed and scalable, using space that is constant per data file. We use lifecycles and opportunity analysis to reason about improved task placement and reduced data movement for five scientific workflows with different characteristics. Case studies show improvements of 15×, 1.9×, and 10–30×. Our work is implemented in the DataLife tool.

CCS CONCEPTS

• **General and reference** → **Performance**; **Measurement**; **Metrics**; • **Information systems** → **Information lifecycle management**; *Distributed storage*; *Hierarchical storage management*.

KEYWORDS

distributed workflows, performance analysis, data flow lifecycles, storage bottlenecks, caterpillar tree

ACM Reference Format:
Hyungro Lee, Luanzheng Guo, Meng Tang, Jesun Firoz, Nathan R. Tallent, Anthony Kougkas, and Xian-He Sun. 2023. Data Flow Lifecycles for Optimizing Workflow Coordination. In *The International Conference for High Performance Computing, Networking, Storage and Analysis (SC '23), November 12–17, 2023, Denver, CO, USA.* ACM, New York, NY, USA, 13 pages.

SC '23, November 12–17, 2023, Denver, CO, USA

ACM ISBN 979-8-4007-0109-2/23/11…$15.00
https://doi.org/10.1145/3581784.3607104

1 INTRODUCTION

Scientific exploration is increasingly distributed and storage intensive. To focus on exploration rather than application development, domain scientists emphasize productivity and flexibility by creating converged workflows that compose applications with different characteristics — e.g., numerical solvers, data analytics, and machine learning (ML) — using storage (I/O) [15]. With large datasets and distributed data sources, the primary performance challenge of workflows is managing data accesses and flows on distributed resources [15, 28, 48, 49].

Many of these bottlenecks can be removed with better workflow coordination, including resource selection, task assignment, and data placement. To guide these decisions, this paper introduces *data flow lifecycle* analysis. Workflows are commonly represented using task DAGs, or directed acyclic graphs where vertices are tasks and edges show ordering dependencies [9, 13, 23, 50, 54, 59, 65]. Data flow lifecycles (DFL) enrich task DAGs with data objects and properties that describe data flow—accesses, reuse, volumes, footprints, and rates—and how tasks interact with that flow. DFL graphs enable analysis from several perspectives: *task*, *data*, and *data flow*. Each is important for identifying potential bottlenecks and opportunities. The lifecycle graphs can be formed from either static or dynamic workflow analysis.

Understanding DFLs can identify and suggest performance improvement opportunities that task DAGs cannot. For example, our analysis identifies the following: (1) large data volumes; mismatched data rates; (2) data non-use, or when data is unused by consumers; (3) intra-task spatial-temporal locality; (4) inter-task data locality, or when data is used by multiple tasks (5) critical and non-critical data flows (6) trade-off between task and data parallelism.Table 1 gives a more complete list.

The opportunities identified from the DFL graphs inform remediation strategies. They can include intelligent task and data placement that pair tasks and flow resources, distributed caching to reduce unnecessary movement, anticipatory data movement to avoid stalling, and write buffering to move flow off the critical path. Often a single strategy applies to multiple opportunities; and some strategies may be easier to enact than others.

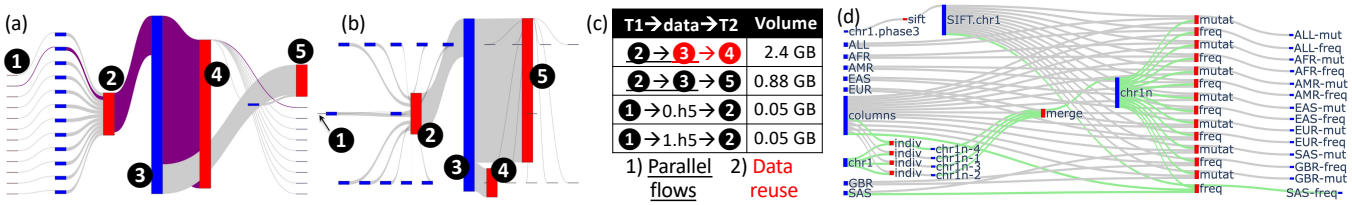


Figure 1: Workflow data flow lifecycles (DFL) (a) capture data flow as a graph, where flow between tasks (red) and data (blue) is left-to-right. Automated DFL analysis identifies remediation opportunities (b) using DFL caterpillar trees, a generalization of critical path, and (c) ranks them by importance. (d) DFL analysis uses novel notions of caterpillar trees, including one showing a path's data branches and task joins.

Although data flow analysis is a well known technique for program compilation and execution [55, 58, 74], there have been few applications to workflows. Data flow analysis is based on static program analysis and may include dynamic disambiguation. The methodology cannot easily be applied to workflows because of different programming models, data accesses across different abstraction layers (memory vs. storage vs. network), and input-dependent behavior. Recent work has recognized that workflow DAGs can be enriched with data nodes to improve scheduling [21]. However, it assumes a single and uniform set of data accesses. Our work generalizes the DAG representation to dynamic flow and introduces several lifecycle relations (patterns) and properties (including data reuse, rates, volumes, footprints, parallelism).

This paper overviews our approach (§2); describes distributed measurement (§3); defines lifecycle representation, properties, and entities (§4); and presents automated analysis that highlights promising opportunities for performance improvement (§5). Our work is implemented in the DataLife tool [44]. We evaluate (§6) our approach and tools on five different workflows. We show how to use our opportunity analysis to reason about converting a workflow pipeline designed for throughput into one targeting response times. Increasingly, optimizing for response time is important because it enables the real-time decisions needed for instrument control, experimental feedback, and automating scientific discovery. Our three case studies demonstrate execution time improvements of 15×, 1.9×, and 10–30×.

Our contributions are as follows:

- Definition, measurement, visualization, and analysis of data flow lifecycle graphs for distributed scientific workflows.
- Scalable monitoring to capture components of data flow lifecycles, where the size of DFL measurements is proportional only to task-file instances and is constant per data file.
- Methods to construct, analyze, and identify opportunities for improving workflow recomposition, task assignment, and data flow. The methods are efficient, i.e., linear in edges and vertices.
- Evaluation of data flow lifecycles with contrasting workflows and use cases that demonstrate workflow performance improvement of 15×, 1.9×, and 10–30×.

2 OVERVIEW

Data flow lifecycles (DFL) enrich task DAGs with vertices, edges, and properties that describe data flow and how tasks and data objects interact with that flow. The result takes the form a DFL DAG (DFL-DAG), which can be generalized as graph (DFL-G) if, e.g., some task nodes are merged. Although a DFL-DAG can be formed from either static or dynamic analysis, our approach focuses on the latter to capture execution properties and precise dependencies. With our approach, the DFL-DAG's *structure* is *independent* of execution, coordination framework, and resource assignment, as long as multiple workflow executions of the same input have the same DAG. An overview of our approach is depicted in Fig. 1. Our work is implemented in the DataLife tool [44].

The first step of DFL analysis is collecting the information needed for DFL graphs. Our method uses runtime measurement to capture data flow characteristics. For this paper, a data object is a file. To avoid the enormous space required by I/O traces, we use distributed and scalable monitoring such that DFL information is proportional only to task-file instances because it is constant per data file.

The next step is forming a DFL graph. To represent an execution's forward flow, we use a *DFL DAG* (DFL-DAG), which is acyclic. Figure 1a shows an example DFL-DAG. Tasks and data objects are vertices; directed edges show data flow. Graph vertices and edges are annotated have *properties* that represent execution and flow, including data accesses, reuse, volumes, footprints, and rates.

To visualize lifecycle graphs, we use Sankey diagrams [5, 40] (Fig. 1a). These diagrams indicate execution and flow properties using scaled vertices and edges.

It can also be useful to aggregate nodes in the DFL-DAG to form a *lifecycle template* (DFL-T). A common example is representing control loops by aggregating parallel instances of the same task. We generalize either DFL-DAGs or DFL-Ts by varying a key input parameter and forming averaged graphs from several executions. Property values are either averaged or represented as histograms.

The final step is identifying opportunities for improving workflow performance. This consists of finding and ranking potential opportunities based on DFL-G structure and patterns.

To narrow the search, we use a generalization of critical path called a *DFL caterpillar tree* (CT). DFL CTs have costs based on execution and flow properties. Figure 1b shows a CT based on flow volume (for subfigure (a)) while Fig. 1d shows a CT based on data producers and consumers.

We then find opportunities by identifying patterns in the critical and near-critical CTs. Patterns are defined by graph *entities*, or relations between vertices and edges. Using CTs, the DFL analysis is linear in edges and vertices.

Finally, we rank opportunities by bottleneck severity. The ranking is based on pattern metrics formed from task, data, and flow properties. Figure 1c shows a ranked table for subfigure (a) that

compares the opportunities (patterns or entities) of given types. The results can be used for recomposing the workflow, improving task assignment, or improving data movement.

3 DISTRIBUTED MEASUREMENT

This section describes distributed and scalable measurement of a workflow's execution for constructing DFL graphs (DFL-G).

Capturing data flow. As DFLs are based on dynamic data flow, the first step is monitoring a workflow's I/O during executions with representative inputs. The monitoring overrides I/O calls using typical dynamic linking techniques e.g., Linux `LD_PRELOAD`. In this paper, we focus on (serial) POSIX and C I/O, which includes all variants of `open`, `close`, `read`, `write`, `fseek` etc.

Characterizing data flow. Understanding *flow lifetimes* requires more information than captured with than either I/O caching [10, 62] or I/O tracing [17, 47, 69]. To track flow between I/O operations, we must know when the same datum is accessed. There are two implications. To track data addresses, it is necessary to maintain the state of the I/O stream that is opaque in I/O operations like `read` and `write`. These operations take an opaque handle and depend upon its state to know what data to access. Second, to maintain histograms that map distinct data items to flow metrics, data identities must be understood at run time, ruling out postmortem analysis. We therefore track the state of each opaque I/O handle (file descriptor or I/O stream). We shadow each I/O handle and maintain its state by intercepting and emulating the effects of all relevant I/O operations.

A task-file pair represents data flow. For each task-file pair, we maintain a histogram of data accesses and access statistics. A single histogram represents one or two flow relations, either *producer* or *consumer*, depending on operation type. I/O reads show flow from data to a consumer task. Writes show flow from a producer task to its data. A consumer relation is a DFL-G directed edge from a data to task vertex; a producer is the reverse. Over multiple tasks and files, many consumer and producer relations are formed corresponding to DFL-G edges. The DFL-G is built by connecting all edges.

A given histogram maintain statistics for each data block in a file. The statistics include operation type, access frequency, distance, and rate. From these statistics, we form DFL-G *properties* such as data volume, block access frequency, access rate, etc. that annotate edges and vertices.

Scaling. To scale DFL-G measurement, each histogram's size should be *constant* rather than proportional to either operations (as in tracing) or file sizes (as in naive histograms). If constant-sized, the total size of all workflow histograms is proportional to instances of task-file pairs, which is typically very modest.

The size of a histogram is the product of data locations and statistics. The number of statistics is a constant (bounded by $\approx$10). We employ two methods to bound the number of locations: adjustable access resolution and spatial sampling. The maximum number of locations is determined by access resolution or blocks per data file. The block size (in bytes) is selected based on expected of data volumes. For reads, block size is a ratio of file size; for writes we use historical information or user guidance.

The second method, spatial sampling, can be used to additionally reduce data locations. Spatial sampling selects a representative fraction of the data addresses and tracks lifecycles for them. Note that sampling is on data locations and not I/O operations

To ensure correct tracking, *the selected fraction of addresses must be the same for each producer and consumer in a lifecycle.* We design a sampling rule that, given a location, determines whether to track it or not. The correctness requirement means that the *rule must be deterministic and depend only on location*, i.e., *independent* of access order and the volume of the data accessed. To sample lifecycles (connected flows), we adapt a strategy designed for single flows [68]. Given a deterministic hash function H, we track a location L if

$$H(L) \bmod P < T$$

using modulus P and threshold T. Each sample represents $1/r$ locations where the sampling rate $r = T/P$. For example, setting $P = 100$ will sample roughly T percent of the whole space.

When combining both strategies, histogram size is effectively bounded by a constant, establishing the total size bounds above.

4 DATA FLOW LIFECYCLE GRAPHS

This section defines DFL graphs and discusses methods for construction (§4.1), analysis (§ 4.2 and 4.3), and visualization (§4.4).

4.1 Lifecycle graphs, DAGs, and templates

Once lifecycle measurements are gathered, a *DFL graph* (DFL-G) can be constructed. Since measurement histograms capture all graph edges, the DFL-G is built by connecting all edges. DFL-G construction can be parallelized by ensuring vertex updates are atomic. However, edge sizes are typically modest.

A DFL-G is a *property graph* [56], where both vertices and edges are annotated with property values formed from histogram statistics. Recall that both tasks and data files are vertices in the graph. A read operation (by a consumer task) is denoted by a directed edge from a data source (vertex) to a task sink. A write operation (by a producer task) is denoted by a directed edge from a task source to data sink.

We denote the vertex sets for data files and tasks with D and T, respectively. Then, the vertex set is $V = \{D \cup T\}$. The edge set is

$$E = \{(u, v, p) : (u \in D, v \in T) \text{ or } (u \in T, v \in D), p \in \mathbb{R}\}$$

where each edge (u, v) is annotated with a lifecycle property value p. In practice, p is a set of properties per edge and vertex.

Connecting edges creates an acyclic graph since each task instance is a distinct vertex. We call this graph a *DFL-DAG*. It can also be useful to aggregate instances of the same task vertex to create a *lifecycle template* (DFL-T), potentially forming cycles.

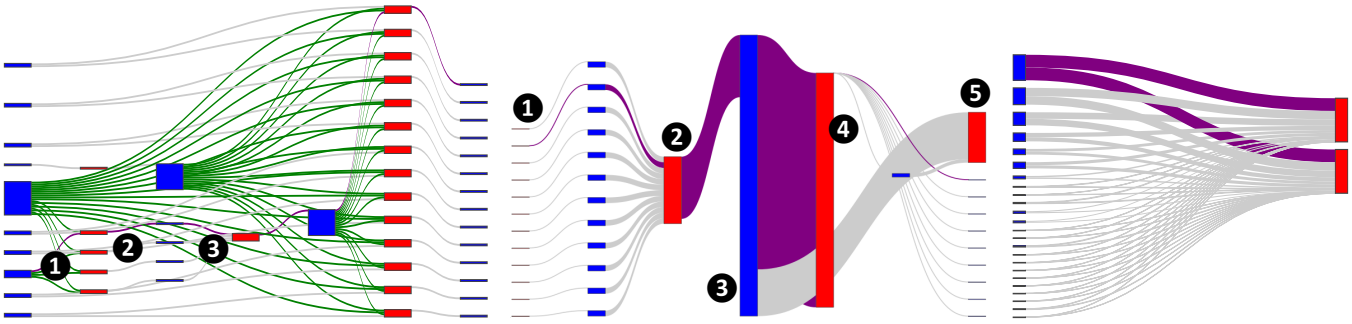
By itself, the DFL-G is cumbersome to manually analyze. The following subsections discuss techniques for analyzing graph entities, and visualization. The next section (§5) discusses opportunity analysis using DFL-G patterns.

4.2 Lifecycle properties

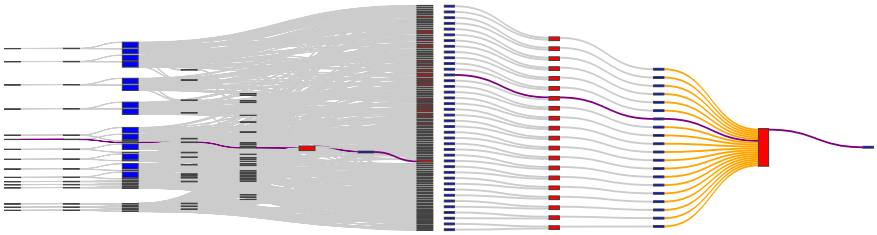
To understand dynamic data flow, we convert histogram statistics into several classes of lifecycle *properties*.

Base properties form the core metrics.

- *Task lifetime*: task execution time
- *File lifetime*: time between open and close
- *Access frequencies* for each data block



(a) 1000 Genomes (b) DeepDriveMD (DDMD) (c) Belle II Monte Carlo



(d) Montage (e) Seismic Cross-Correlation (Seismic)

Producer→data→consumer	Volume
aggregate (❷) → a.h5 (❸) → train (❹)	2.4 GB
aggregate (❷) → a.h5 (❸) → lof (❺)	0.88 GB
openmm (❶) → 0.h5 → aggregate (❷)	0.05 GB
openmm (❶) → 1.h5 → aggregate (❷)	0.05 GB

(f) Ranking of DDMD composite edges (top 4).

Figure 2: (a)-(e) Data flow lifecycle DAGs for five workflows. (f) A producer-consumer pattern ranking for DDMD. Data flow is left to right. Tasks and data are red and blue, respectively. Critical paths are purple. For DDMD, Belle II, and Montage, critical path shows data volumes; for 1000 Genomes, it shows path with most instances of data branches > 2 (green) and task joins; for Seismic, it shows most instances of task fan-in (yellow).

- *Data volume*: total volume (non-unique) of data
- *Data footprint*: unique volume of data
- *Read (write) latency*: total read (write) latency

Ratios capture measures for data rates and intensities.

- *Read (write) rate*: ratio of read (write) *operations* to task time
- *Data read (write) rate*: ratio of read (write) volume to task time
- *Read (write) blocking fraction*: fraction of time for read (write) blocking time during an open I/O stream

Access patterns describe lightweight data access pattern statistics for a specific data object.

- *Consecutive access distance*: linear ("seek") distance between consecutive access locations i and j for spatial locality.
- *Reuse* and *subsets*: degree that data is reused or unused.
- *Use concurrency*: whether data is accessed by many tasks.

Opportunity analysis (§5) finds higher-level workflow patterns.

Figure 2 shows examples of reuse and concurrency for two workflows. (Detailed discussion is deferred to §6.) Figure 2a shows single-use concurrency in 1000 Genomes workflow [1]. The workflow processes multiple chromosomes. The biological signature of each chromosome is specified in a file. Each file is processed by multiple `indiv` by tasks, concurrently and independently, where each task processes a disjoint chunk. The DFL-G shows the concurrency via edges (❶) emanating from each chromosome file (data vertex) to its corresponding `indiv` task (task vertex). The pattern is also data parallel, important later, since each chunk is read once and processed by a separate task.

Figure 2b shows two kinds of reuse in DeepDriveMD (DDMD) [45]. In this workflow, a `simulation` stage (❶) generates several HDF5 files that an aggregator (❷) combines (❸). The incident edges of this combined file show much larger data volume outflow than inflow, implying data reuse. (The reuse can be seen by zooming into edges per file block.) The downstream task ML `train` (❹) repeatedly accesses data from the combined dataset (❸), resulting in *intra*-task data reuse. The downstream task `lof` (❺) reads the same data, resulting in *inter*-task reuse.

4.3 Perspectives, entities, and ranking

DFL-Gs enable flow analysis from several perspectives:

- *task*: what data a given task produces or consumes
- *data*: what tasks produce or consume a given data file
- *flow*: what data movement is required for execution

To facilitate analysis, we identify the following lifecycle *entities*, which include both graph constructs and relations between them:

- *data* or *task* vertex;
- *data* or *task relation*: a task (data) vertex and its incident edges;
- *producer* or *consumer relation*: the *simple* edges between a producer task and a dataset, or the reverse;
- *producer-consumer* or *consumer-producer relation*: *composite* relation from a producer, dataset, and consumer; or the reverse.

To quickly highlight entities of related to bottlenecks or opportunities for improvement, we associate property values with entities. We then form a projection of the entity according to a selection criterion and rank order the results by property values.

An entity *projection* refers to the process of extracting an entity type from the original DFL-G. A *ranking* refers to sorting the projected set by property values. As a simple example, data vertices can be sorted and ranked by data volume or rate to identify which

Pattern	Definition	Identification in DFL-G	Remediation
Data volume	Tasks read/write large data volumes	DFL-G *flows* with volumes exceeding storage or network ability	Pair tasks & storage resources; write buffering; anticipatory data movement
Mismatched data rate	Mismatch between rate of data production and data consumption	DFL-G *flows* with data rates that cause stalls, likely on critical path	Pair tasks & flow resources; adjust data generation rate; data filtering/compression.
Data non-use	Data not used by consumer in whole or part	DFL-G *data* leaf vertex: data unconnected to consumers, DFL-G *flows* with total data transfer smaller than the data file size.	Selective movement, e.g., on-demand caching; data filtering.
Intra-task data locality	Spatio-temporal access locality within a file	DFL-G *data* vertex with consecutive access distances < block size; 0 indicates temporal locality	Caching (including hints and biased policies); block prefetching.
Inter-task data locality	Same data is used by multiple tasks or task instances.	1) DFL-G *producer* & *consumer* write/read same *data*. 2) DFL-G *task* accesses same *data* over multiple instances (e.g., loops) 3) DFL-G *data* read by multiple *consumers*.	1) Caching, co-scheduling. 2) Data retention; caching. 3) Co-scheduling and data placement.
Critical (non-critical) data flow	Data flows that must improve (or could relax) to improve response time (to free resources)	1) DFL-G's *caterpillar* has *flow* that causes stalling (critical). 2) DFL-G *producer* and *consumer* use same data and *consumer* can proceed without all inputs. [Must validate (2)]	1) Bias resources for critical vs. non-critical tasks & flows; anticipatory data movement. 2) Change task-data synchronization (e.g., pipeline data push/pull).
Task and data parallelism trade-off	Trade-off between task parallelism (for better response time) and overhead of tasks or data flow (e.g., increased I/O contention, rates).	The *in-degree* (number of neighboring *data* vertices) of a *consumer* task in DFL-G. The in-degree implicitly specifies number of *producer* task vertices executed concurrently. [Must validate]	Coordinate parallelism, task placement, and data flow resources.

Table 1: Identification of opportunities in the DFL DAG and potential remediations.

data files should be prioritized for storage and flow resources. Figure 2f ranks the DDMD workflow's producer-consumer relations by volume. Ranking provides an important way to identify the lifecycle entities most likely to benefit from techniques such as recomposition, co-scheduling or task-data co-location. Opportunity analysis will be discussed in more detail in §5.

4.4 Visualizing lifecycles

To visualize DFL-Gs, we use Sankey diagrams [5, 40], a representation designed to show flow from one set of items (vertices) to another. We extend the typical diagram of two sets to apply to both DFL-DAGs and DFL-Ts. Figure 2 shows examples for a diverse set of workflows. Each vertex is represented as a rectangle and flow between pairs of vertices is represented with a directed edge. The width of the edge is proportional to a selected property, showing the significance of the flow.

5 INSIGHT AND OPPORTUNITIES

A primary purpose of DFL graphs is to facilitate identification of potential opportunities for improving the overall performance of a workflow. We now describe automated opportunity analysis to focus an analyst's attention on a small set of candidates. The opportunity analysis is efficient, i.e., linear in edges and vertices.

Table 1 summarizes the identified opportunities. The patterns are based on a combination of the entity analysis previously described (§4.3) and higher-order relations introduced in this section. The table lists several possible remedies, where the best candidate depends on workflow coordination and available resources and therefore requires user direction. Some opportunities require human validation to ensure correctness, indicated by "[Must validate]".

One method for identifying patterns in the DFL-G is graph pattern matching [20], i.e., finding all subgraphs that are isomorphic to each pattern. However, the search space with these approaches can grow exponentially with pattern size and is NP-complete [29].

In contrast, our methods have a worst-case complexity of linear in edges and vertices. The basic idea is to leverage DFL-G structure, characteristics of target patterns, and domain knowledge. The first step narrows the opportunity search (§5.1). The second identifies target patterns using only highly local knowledge, primarily a vertex and its incident edges (§ 5.2 to 5.4). This section shows how to detect each pattern, indicates their potential opportunities, and suggests potential remediation strategies.

5.1 Narrowing the opportunity search

Two popular methods for partitioning graphs for resource assignment are min/max flow (e.g., communication) [26, 34, 41] and acyclic hierarchical partitioning (e.g., circuits) [36, 51].

We adopt a different method based on an extended notion of critical path. The advantage of our method is that it highlights opportunities that are more likely to improve workflow *response time*. The reason is that our method prioritizes graph subsets with a dependent series of tasks where most producer-consumer exchanges have high locality. Thus, in contrast to the above methods, it tends to find threads of execution where coordination can be localized.

To narrow the opportunity search, we rely on two graph theoretic concepts: critical path and caterpillar tree. A *critical path* is the longest path in a directed acyclic graph, where vertices and/or edges are weighted with a certain property value. Critical path analysis (CPA) typically finds the minimal time required to finish a workflow. The critical path forces tasks on other paths to wait. Patterns along this path therefore highlight opportunities for shortening the path.

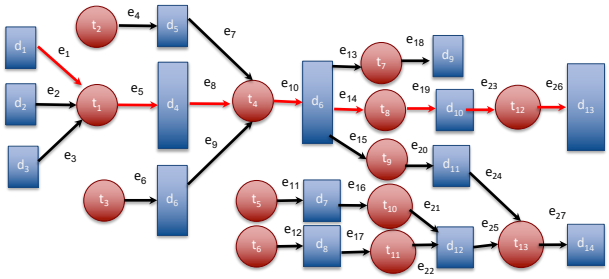
Our analysis performs CPA with respect to several different properties, yielding *generalized critical path analysis* (GCPA). By adopting different properties the path focuses on different bottlenecks. By exploring the the properties footprint, volume, and flow

rate, the analysis identifies potential bottlenecks corresponding, respectively, to storage capacity, transfer volume, and transfer speed. Identifying these alternative bottlenecks is crucial for better task coordination and resource assignment for both data and flows.

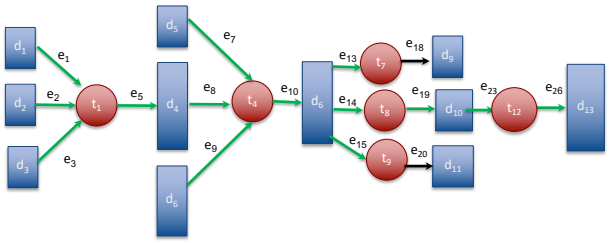
Given a critical path we want to identify patterns along it. Unfortunately, critical paths are too small a graph subset because they may not show all relevant producer or consumer relations for a critical task or data vertex. To address this, we extend the path into a *caterpillar tree*. A *caterpillar* or *caterpillar tree* is defined as a tree in which all vertices are within distance one of a critical path [2, 25, 33]. Caterpillars identify all distance-one fan-in and fan-out tasks on the critical path. Figure 3 shows a DFL graph, the critical path, and its associated caterpillar tree.

Unfortunately, because DFL-Gs have two types of vertices (data and task), a normal caterpillar destroys some relevant producer/-consumer relations. Our *DFL caterpillar* extends the caterpillar with the following rule: if the roots of the caterpillar are data vertices, also include distance two vertices from the critical path (d_9 and d_{11} in Fig. 3b) that are associated with the producer tasks (t_7 and t_9 in Fig. 3b respectively) as part of the caterpillar tree.

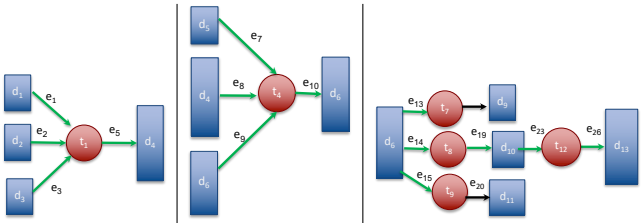
Once the critical paths and associated DFL caterpillars are computed, we identify patterns with respect to either *task relations* or *data relations*. Considering both is important because each relation makes detection of a pattern a matter of only a vertex and its incident edges. Then, the basic optimization strategy is two-fold: a) *parallelize between trees* (threads); and b) *improve execution of critical caterpillar tree fragments* with increased data locality or improved flow resources.



(a) A DFL graph with critical path (red edges). Tasks and data are red circles and blue rectangles, respectively. An edge property e_i depicts metric values.



(b) Narrowing a search with the DFL caterpillar (green edges).



(c) Aggregator: equal in/out-flow. **(d) Compressor: smaller out-flow.** **(e) Composite pattern.**

Figure 3: Identifying opportunities in data flow lifecycles.

5.2 Patterns from data relations

A data relation is a data vertex and its incident edges. We distinguish data relations based on incoming and outgoing edges. There are four categories: *regular* (one in, one out); *fan-in* (many in, one out); *fan-out* (one in, many out); *fan-in/out* (many in, many out).

Data relations identify the following interesting patterns. First consider *single consumer* (one-out) relations. If the consumer utilizes only a subset of the data, it exhibits a *data subset* pattern that will be reflected in DFL-G property values. For example, if total footprint consumed is smaller than the dataset size, only a subset has been accessed. This likely implies unnecessary data movement, either by the producer or by the consumer (data volume > dataset size), or both. If a consumer accesses the same data block multiple times, there is *intra-task data reuse*. In this case, hot blocks can be cached to accelerate same data accesses. If a consumer access pattern has a small consecutive access distance, it exhibits spatial locality and likely benefits from prefetching and caching.

Now consider data relations with *multiple distinct consumer tasks* (fan-out). If the same data is used by multiple consumers it may be profitable to exploit this *inter-task reuse* using caching or co-scheduling. If each consumer processes a partition of the input data, there is *data parallelism*. Consider either a parallel file system (parallel I/O), a filesystem with high metadata throughput (metadata contention with serial I/O), and co-scheduling some consumer tasks to convert disk and network flow into local in-memory flow.

5.3 Patterns from task relations

We distinguish task relations in the same way, resulting in *regular*, *fan-in*, *fan-out*, and *fan-in/out*.

An interesting and common pattern is *aggregator (fan-in) data parallelism* (Fig. 3c). A variant is the *compressor-aggregator with data parallelism* (Fig. 3d).

In the former, several input streams, usually of the same size, are inputs to a task. The aggregator task combines them into a new composite file, possibly also applying a minor transformation. An example is the `aggregation` stage DDMD. In DDMD, either the point cloud or the contact map from separate files are processed and aggregated to a new HDF5 file. Since aggregation likely introduces unnecessary serialization and data movement, the convenience of a single input must be evaluated with this overhead. If convenience is paramount, the aggregation should possibly be pipelined across multiple network links (to avoid network contention) or multiple/improved storage resources (to avoid I/O contention).

In a *compressor-aggregator* pattern, the aggregator additionally compresses data. An example is in 1000 Genomes where the `merge` task combines several input files into a compressed `tar` file. If assigned to the correct resource, this can benefit downstream flows

over slow storage and networks. Alternatively, the compression may not be worth the serialization.

5.4 Patterns from task-data compositions

More interesting patterns emerge when task and data relations on the DFL caterpillar are composed.

We provide three examples. First, a *series* of task and data relations. Consider re-composition and co-scheduling to improve performance. Second, an *aggregator task followed by a regular task*. An example is DDMD (Fig. 2b); consider coalescing or co-scheduling the aggregator and its consumer.

A third category of compositions involves forms of data parallelism and overlaps with the previous category. Examples include combinations of *aggregators* and *splitters*. Figure 3e shows an example of splitter tasks, where one is followed by a regular task. 1000 Genomes shows an example of *aggregator followed by a splitters*: `merge` gathers information from all `indiv` tasks and scatters it over subsequent `freq` and `mutat` tasks.

Several remediation strategies are relevant. If an aggregator's only function is combining producers, it likely introduces unnecessary sequencing. Alternatively, a very large aggregator may benefit from additional subaggregators for locality domains so that network or storage contention can be avoided. In addition, appropriate storage resource assignment can minimize cost associated with the overhead of opening/closing many small files.

Again, the common theme in these remediation strategies is to *improve execution of DFL caterpillar fragments* and *parallelize separable caterpillars or paths*.

6 EVALUATION

This section demonstrates the applicability of data flow lifecycles to distributed scientific workflows. We use DFLs and opportunity analysis to reason about improved task placement and reduced data movement for five scientific workflows with different characteristics. We overview characteristics and opportunities for each workflow (§6.1). Three case studies (§ 6.2 to 6.4) apply different remediations techniques to selected workflows and show performance improvements of 15×, 1.9×, and 10–30×, respectively. In the first two case studies, we optimize workflow for *response time*. The third case is optimized for throughput. Increasingly, optimizing for response time is important because it enables the control loops needed for automating experiments with instruments.

Our evaluation uses the machines in Table 2. Although our evaluation uses compute resources at medium to smaller scales, our results are still value for the supercomputing community. First, these scales are often appropriate for control loops. Second, it is often more difficult to find performance opportunities at smaller scales than large because there is less likelihood for moving massive data volumes and causing resource contention [48].

6.1 Workflow overview

We compare and contrast DFL-Gs for five contrasting workflows. Figure 2 shows the signature DFL-DAGs for each workflows. Figure 4 shows their corresponding DFL caterpillars. For each workflow, we identify interesting patterns and opportunities that can be observed along their DFL caterpillars.

Machine	Compute, Memory	Storage options (notes)
CPU cluster	2× Intel SkyLake; 192 GB	NFS (default); Lustre; SSD (node); Ramdisk (node)
GPU cluster	2× AMD EPYC; NVidia RTX 2080 Ti; 384 GB	NFS (default); BeeGFS (w/ caching); SSD (node); Ramdisk (node)
Data	2× Intel Haswell, 128 GB	Remote storage via 1 Gb/s WAN

Table 2: Machine configurations for experiments.

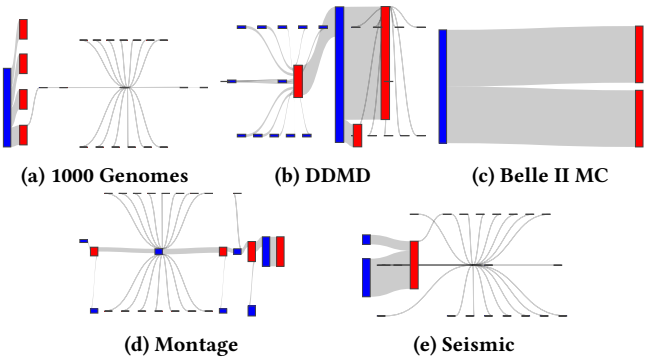


Figure 4: DFL caterpillars for the DFLs in Figs. 2a to 2e. Task and data vertices are red and blue, respectively.

1000 Genomes. Figure 2a shows the DFL-G for 1000 Genomes [1], a data-intensive bioinformatics workflow. The critical path (numbered) represents the path with most instances of *data branching* (fan-out) and *task joining* (fan-in); it is part of the DFL caterpillar tree shown in Fig. 4a. The data file with fan-out (❶) to four `indiv` tasks corresponds to the "multiple distinct consumers" pattern with data parallelism (§5.2). It suggests an opportunity to reduce the pattern's time using increased task parallelism. A more complete picture emerges with the composite producer-consumer pattern represented by ❶, ❷, and ❸. There is a trade-off between task parallelism and I/O contention from processing small files (❷) by the consumer (❸). Our case study (§6.2) creates a pipeline optimized for response time (vs. throughput) and adjusts parallelism and assignment of tasks and data flow resources (Table 1), improving performance by 15× (§6.2).

DeepDriveMD. Figure 2b shows the DFL-G for DeepDriveMD, the deep learning-driven molecular dynamics simulations workflow for protein folding [3, 45]. DFL-G edges and critical path represent *data flow volume* or the path with the most data movement. Figure 2f ranks the producer-consumer relations by volume. Observe that both `train` (❹) and `lof` (❺) uses the same data (❸). However, `train` accesses far more data (2.4 GB vs. 0.88 GB)—*and more than is produced by* `aggregate` (❹). Thus, `train` has substantial intra-task locality through data reuse and would benefit from caching. DFL-G analysis using footprint (not shown) reveals *data non-use*, i.e., *only half the data produced by* `aggregate` *is used by both consumers*. As discussed in our case study (§6.3), these facts also point to a trade-off between (a) work minimization—one `aggregate`—that introduces serialization and (b) work replication—parallel `aggregates`—that increases task-data parallelism. If there are available data flow resources that can be enabled with intelligent task-data coordination, the latter choice reduces response time. Our case study leverages

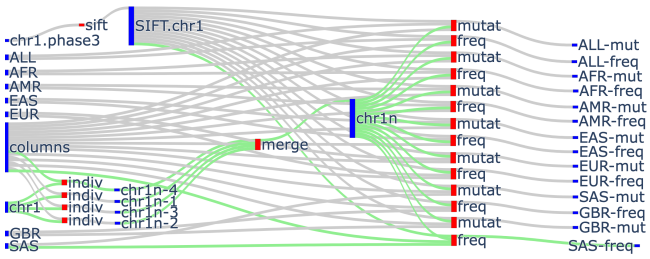


Figure 5: 1000 Genomes: DFL caterpillar (green) by data branches and task joins. Tasks are red; data blue.

this and other insights to create a new pipeline optimized for response time (vs. throughput) and improves it by 1.9× (§6.3).

Belle II Monte Carlo. Figure 2c shows a reduced but representative DFL-G for Belle II MC, a data-intensive physics workflow [32]. Edges represent data volume. The DFL-G shows recurring data-task fan-in from a single collective dataset. The pattern immediately highlights inter-task data reuse, i.e., multiple tasks using the same file. The DFL-G's consecutive access distance property (not shown) reveals many access distances of 0, indicating intra-task spatial locality. To exploit the first opportunity, tasks could be scheduled at the same time on nearby resources and use distributed caching. Caching and data prefetching can address the second. Our case study (§6.4) demonstrates an impact of 10× from caching.

Montage. Figure 2d shows the DFL-G for Montage [4, 57], a compute-intensive astronomical image processing toolkit that uses a collection of structural images to build a composite image mosaic for viewing outer space. To assemble a large number of small image files, it re-projects the images through a common frame to match the background and determine overlaps, generating many intermediate files for correction and adjustment. The computational component results in low effective data rates and low I/O operations. Consequently, there is room to parallelize or accelerate tasks without overburdening flow resources.

Seismic Cross Correlation. Figure 2e shows the DFL-G for Seismic Cross Correlation [6, 27], a data intensive workflow that computes cross correlation of signals from multiple seismic stations, identifies good fits, and compresses them in a single file. The critical path represents instances task fan-in (joins). The DFL-G's overall pattern is a multi-stage aggregator, that suggests potential for a trade-off between parallelism vs. data and flow locality. On one hand, a multi-stage aggregation could introduce more task and flow parallelism; and near-data operations could reduce the downstream flow to avoid resource contention. On the other hand, reducing the stages with task composition would reduce data movement and increase locality.

6.2 Case study: 1000 Genomes

The 1000 Genomes project [22] studies the genomes of 2504 individuals from 26 populations to understand human genetic variation. The 1000 Genomes proxy application [1] has five tasks: `individuals` (chromosome processing), `individuals_merge` (aggregation), `sifting` (Single nucleotide polymorphisms (SNPs) variant SIFT scores), `frequency` (overlapping mutation frequency), and `mutation_overlap` (overlapping mutations). In subsequent discussion, we abbreviate the five tasks respectively as `indiv`, `merge`, `sift`, `freq`, and `mutat`. The default configuration targets throughput performance with large chunks of work but very long run times. We are interested in *efficient* response time.

DFL-G analysis. Figures 2a and 5 show DFL-Gs. As previously observed, there is an opportunity to increase task parallelism. However without careful task and flow coordination, it increases data movement and contention and degrades response time.

Figure 5 shows the DFL-G for Chromosome 1 (`chr1`). There is a dependency between `indiv` and `merge`; `sift` is independent; and `freq` and `mutat` are consumers of the output files generated by `merge` and `sift`. In addition, there are many explicit data branches (fan-out, green). For example, 1) `columns` and 2) `chr1` are consumed by multiple `indiv` tasks. The data branches indicate *duplicated* and *congested* data flow as the same data is consumed by many tasks at once. Either staging or caching would increase locality of data and consumer tasks and alleviate contention.

DFL caterpillar. Figure 5 and Figure 4a show the Sankey-based and simplified caterpillar, respectively, based on instances of data branches and task joins. There are five branches and four joins (aggregation on `indiv`, `merge`, `sift`, and `mutat`). Recall the caterpillar tree expands the critical path to include all significant tasks and data: all `indiv`, `merge`, `sift`, and `mutat` tasks and their inputs. We identify 10 caterpillar trees, one for each chromosomes. Because (a) the tasks along each caterpillar have several dependencies and (b) there are limited dependencies between caterpillar trees, *we increase the locality of tasks and data within each caterpillar.*

Parallelism and staging. To maximize producer-consumer tasks locality, we schedule tasks within a caterpillar tree to the same compute node and stage their input files (`columns` and `AMR`) to the *much faster* node-local file systems provided by an SSD and RAM-disk. We co-locate the `sift` task to run concurrently with `merge` for better parallelism. With that, the execution is partitioned into *four* stages: **stage 1** for data staging that moves input files from parallel file systems such as BeeGFS to local storage; **stage 2** includes `indiv`; **stage 3** includes `merge` and `sift`, running concurrently; **stage 4** includes `freq` and `mutat`. The four stages follow the fork-join parallelism. All intermediate files are staged locally.

Evaluation. To demonstrate the efficiency and effectiveness of the proposed design, we perform a comparison study using 6 distinct configurations for I/O staging and distribution. The input problem size is 30, limiting the `indiv` tasks per chromosome to 30. For 10 chromosomes, there are 300 `indiv` tasks, 10 `merge` tasks, 10 `sift` tasks, 70 `sift` tasks and `mutat` tasks. The configurations are:

(1) **15/bfs.** The tasks are scheduled on 15 nodes with SLURM, where the 300 `indiv` tasks of **stage 2** are partitioned into 15 sets, each set handled by one node, followed by **stage 3**, and finalized by **stage 4**. All files and intermediate files are staged in the parallel file system (BeeGFS).
(2) **10/bfs.** In contrast to **15/bfs**, the `indiv` are scheduled on 10 nodes, where 300 indiv are partitioned into 10 sets. Each set of 30 `indiv` is processed on one node.
(3) **10/bfs+shm.** In contrast to **10/bfs**, the intermediate files are staged in local RAM-disks.
(4) **10/bfs+ssd.** In contrast to **10/bfs+shm**, the intermediate files are staged in SSDs.

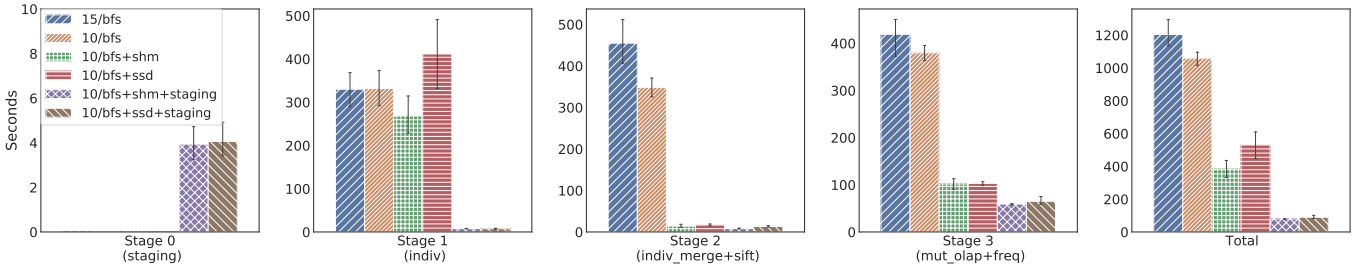


Figure 6: 1000 Genomes: Execution time for each configuration, showing times for each stage and entire workflow.

(5) **10/bfs+shm+staging.** Here **stage 1** opts in and moves all the necessary inputs to tasks to faster, local RAM-disks.
(6) **10/bfs+ssd+staging.** In contrast to **10/bfs+shm+staging**, the necessary inputs are moved to on-node SSDs.

We perform each experiment on GPU cluster (Table 2, using only CPUs) five times and show results in Fig. 6. The distribution on 10 nodes outperforms that on 15 nodes, and staging intermediate data in local storage performs better than staging all files in BeeGFS (*up to a 2.8× speedup*). Staging/moving the input files to local storage leads to a performance improvement (*up to a 6.7× speedup*). Overall, performance *improves by 15×* compared to the original configuration on 15 nodes. This reveals the usefulness of the proposed DFL-G and patterns for tasks and data coordination.

6.3 Case study: DeepDriveMD

DeepDriveMD is a deep learning-driven molecular dynamics simulations workflow for protein folding [3, 45]. It consists of a four-stage pipeline. The default configuration targets throughput performance with large chunks of work and long run times. We are interested in *efficient* response time.

DFL-G and caterpillar. Figure 2b shows a single iteration of the 4-stage pipeline, consisting of simulation tasks (❶), aggregator (❷), training (❹), and inference (❺). Figure 4b shows the caterpillar tree. The critical path is based on data volume. First, DFL-G analysis (Fig. 2b) shows that training consumes 62% of the total data volume of the pipeline. Observe that the incident edges of the aggregated file (❸) exhibit a significantly larger data out volume as compared to the data in volume, indicating intra-task data reuse. Second, DFL-G analysis using footprint (not shown) reveals data non-use; only half the data produced by `aggregate` is used by both consumers. Third, observe that training and inference use the same data. This suggests a trade-off between (a) one aggregation (work minimization) with more serialization and (b) composed aggregations that duplicate work but that occur in parallel and eliminate a pipeline stage. However, the potential is muted because inference relies on training. We therefore explore an alternate pipeline where training asynchronously applies model updates. The new pipeline *co-schedules segments of the caterpillar.*

Pipeline optimization. To improve response time, we increase pipeline parallelism and reduce unnecessary data movement. We coalesce the aggregation with both training and inference to exploit data non-use, reduce data movement, and increase stage parallelism. Additionally, we remove the task-data dependency between training and inference with a *nested pipeline* that trains asynchronously.

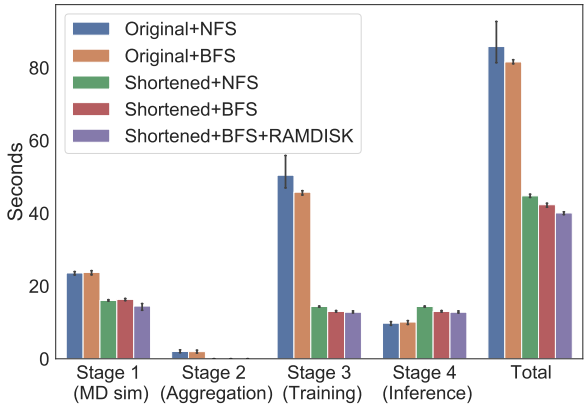


Figure 7: DeepDriveMD: Execution time for Original and Shortened pipelines, showing times for each stage and entire workflow pipeline.

The inner loop is a shortened 2-stage pipeline that excludes aggregation and training. The outer loop includes training, which asynchronously gathers new inputs and produces outputs. For most use cases, this approach will not substantially affect the scientific quality of DDMD's results.

Evaluation. We evaluate two pipelines. 'Original' respects the original synchronization and uses the 4 original stages. 'Shortened' is the asynchronous version with 3 total stages and a 2-stage inner loop. For response time, we use a small time-step size (100) and execute the workflow for 5 iterations. We perform each experiment on GPU cluster (Table 2, using both CPU and GPU) using 2 compute nodes and 12 simulation tasks.

We evaluate five configurations. 'Original' stages through NFS and BeeGFS. 'Shortened' additionally explores RAM-disks because aggregation is now localized. We perform each experiment 3 times. Figure 7 shows results. 'Shortened' shows the impact of a) co-scheduling training and inference and b) local RAM-disk aggregation and results in a *speedup of up to 1.9×*. In 'Shortened', BeeGFS (better I/O parallelism) provides an additional 5.4% improvement, while incorporating RAM-disk with BeeGFS resulted in a further 9% improvement.

6.4 Case study: Belle II Monte Carlo

In the Belle II experiments [32], Monte Carlo (MC) simulations account for well over 50% of computation time [35, 52].

DFL-G analysis. Figure 2c shows a simplified DFL-G. Each task executes several MC simulations, where each simulation uses

Scenario	Pattern	Ensemble	Filter
S1	real	no	no
S2	regular	no	no
S3	real	4×	no
S4	regular	4×	no
S5	regular	no	4×
S6	regular	4×	4×

Table 3: Belle II scenarios.

Cache	Device	Cache Scope	Size
L1	DRAM	Task private	64 MB
L2	DRAM	Node-wide	16 GB
L3	SSD	Node-wide	200 GB
L4	file system	Cluster-wide	512 GB

Table 4: Cache configurations for TAZeR.

between 20–24 input files. There are two key opportunities. The first is intra-task spatial locality due to small consecutive access distances. The second is inter-task file reuse. In a MC campaign, this reuse is dynamic and random. Reuse probabilities can be estimated using a statistical model and knowledge of the number of tasks that draw from set of input files [28].

Both opportunities suggest distributed I/O caching: per-task caching captures intra-task locality, while shared caches (node-wide, cluster-wide) capture inter-task reuse. We evaluate both caching and additional data access optimizations requiring code changes.

We use an I/O intensive configuration of 16 datasets per task, resulting in an average I/O rate of 1600 MB/s. To mimic a typical campaign's distributed environment, tasks run on the CPU cluster and data is served from Data server (Table 2). The workflow is scaled down to execute on 10 nodes, with tasks being allocated to every core (24 cores/node), yielding 240 concurrent tasks.

Distributed caching. We evaluate distributed caching against the typical practice of copying (FTP) each data set before task launch. The caching redirects network and disk operations to cache when possible. We use the TAZeR [28, 62] infrastructure configured with four levels: task-private DRAM, node-wide DRAM, node-wide SSD, and cluster-wide filesystem, specified in Table 4. Distributed caching *improves the execution by a factor of 10.0×*.

Emulated optimizations. To explore additional optimizations, we capture real traces, adjust the traces by how each optimization would affect data accesses, and replay them using the BigFlowSim emulator [28]. The emulator replays all data accesses using real data but simulates compute. We consider six scenarios, a Baseline and 5 combinations of three optimizations. The first optimization regularizes access patterns by 'defragmenting' to increase spatial locality. The second improves I/O locality with task *ensembles* that group 4 tasks per dataset. The third improves I/O locality by converting data field selections into a near-storage *filter* that reduces transferred data by a factor of 4. Thus, ensembles reduce the number of schedulable tasks while filters affect transferred data. The six resulting scenarios are defined in Table 3.

Figure 8 shows the results of the 6 scenarios from Table 3. The figure shows (a) a breakdown of execution components and (b) a line indicating relative execution time (secondary axis). Time 0 corresponds to the time of Scenario 6 with all data staged locally

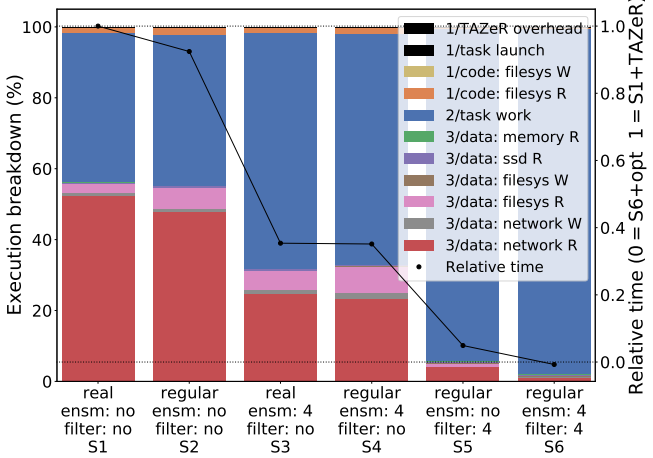


Figure 8: Belle II Monte Carlo: Execution breakdown (bars) and relative time (line) for each scenario (Table 3). The breakdown distinguishes between overheads of TAZeR, transfer of code, and transfer of data, where the latter is divided into several categories.

('optimal'); 1 corresponds to S1 using TAZeR. The emulation is conservative, i.e., provides a lower bound for workflow performance improvements by keeping compute times constant.

Individually, regularized accesses (S2) and ensembles (S3) yield a 6% and 65% improvement in execution time, respectively. Combined (S4), the improvement is 67%. With filters and regularized access (S5), improvement is 95%. With all three (S6), perhaps unrealistic, improvement is ≈100%. *Using the most plausible scenarios (S3, S4), we conclude an additional 2.9–3.0× over 10.0× is possible.*

Execution breakdown. The optimizations affect execution components in different ways. Ensembles (S1 vs. S3; S2 vs. S4) mostly affect the network read portion of data accesses, improving it by about 2×. Compare with a nominal potential of 4× (4 tasks/ensemble). Filters (S2 vs. S5) drastically affect all components of data access time, improving them by a total of 10×. Compare with a nominal 4× (the filtering factor). Improvements can be larger than nominal because although total data read at the Data server stays constant, 4× less data is transferred by it. Then, a larger fraction of each task's footprint remains in cache, vastly increasing accesses in cache levels 1 and 2 (private and shared memory, respectively). Scenarios S5 and S6 show that super-linear speedup are possible when workflow data locality and reuse are improved.

7 RELATED WORK

Static vs. dynamic data analysis. Data flow analysis is a well known technique for static program analysis, program compilation and programmer feedback; cf. [55, 58, 74]. Data flow distinguishes between control and data dependencies, where the former represents dependencies due to conditionals and latter data use. The primary limitation of static analysis is ambiguous data references. In workflows, ambiguity results from input dependencies and accesses across different abstraction layers (memory vs. storage vs. network) and programming models. Thus, we use dynamic data flow analysis with lightweight measurement.

Distributed workflow representation and coordination. Most workflow managers represent workflows using task DAGs that do not represent data objects [9, 13, 23, 50, 54, 59, 65]. They therefore coordinate workflow execution using dependencies that are a conflation of control and data dependencies. Workflow languages such as Parsl [12] and Swift [72] represent distributed tasks using parallel constructs but do not explicitly reason about data flow. The distributed scripting language Intrepydd [78] uses *human* annotations to reason about data flow. In situ frameworks [11, 42, 43, 46, 61, 67] represent workflows with more precision by distinguishing between producer-consumer and publish-subscribe coordination models. However, data is still not represented as a first class citizen. Domain-specific coupling frameworks [30, 60] can generate coupled PDE solvers based on method-specific decomposition schemes rather than explicit representation of data.

The closest prior work on workflow representation is DFMan [21], which schedules workflow DAGs, enriched with data nodes, on HPC systems so as to optimize storage bandwidth. Sun's work [63] adopts a similar goal for coupled applications, but does not focus on representation. Although DFMan represents producer-consumer relations, it assumes a single, uniform pattern of data accesses, which means there are no distinctions between relations, either statically or dynamically. Further, the user must provide the DAG. Our work generalizes the DAG representation to static or dynamic flow for each producer-consumer relation, introduces several new properties (including data reuse, rates, volumes, footprints, parallelism). We also contribute methods for scalable measurement and automatic opportunity analysis.

Some HPC task managers use performance feedback to refine critical path analysis and improve scheduling [8, 64]. However, its representation is still task based and therefore cannot capture data flow. It also requires user-inserted source-code annotations to initiate performance monitoring.

Workflow characterization. Workflow tools fail to capture data reuse and locality over logical lifetimes. Workflow simulators [16, 18] can characterize workflow manager overheads from managing dependencies, packaging, and queuing tasks [19] but do not reason about how to reduce data movement costs through better partitioning or locality. There has been substantial work on caching and staging for storage [7, 10, 13, 38, 39, 46, 61–63, 66, 70], including detailed access and reuse analysis [14, 31, 37, 68, 73, 75–77]. However, the key challenges are still ensuring task-data locality and knowing *what* to cache and *when*.

I/O characterization and monitoring. Several tools monitor and characterize I/O and storage system behavior. I/O tracers such as Darshan [17] and Recorder [69] capture an application's I/O operations [69]. I/O mining tools such as TOKIO [47] and IOMiner [71] analyze collected traces or system I/O logs. Some of these tools analyze I/O operations from the perspective of storage system architecture [17, 47] while others diagnose performance problems [71]. Because these tools focus on I/O operations rather than data lifetimes, they do not track data flow between the I/O operations.

Several studies characterize I/O operations from the perspective of an entire supercomputer. One study [53] analyzes access and reuse patterns for storage data (files) on a production-scale supercomputer; another characterizes I/O behavior of HPC workloads [24]. Neither study captures reuse or data movement between applications or workflow tasks.

8 CONCLUSIONS

We have introduced data flow lifecycles and described methods for representation, measurement, analysis, visualization, and opportunity analysis. The methods are efficient. Measurement data is independent of data volumes, accesses, and time and proportional only to task-file instances. Opportunity analysis is linear in DFL graph edges. Our evaluation compared DFLs on contrasting workflows, highlighted opportunities for remediation, and demonstrated performance improvement of of 15×, 1.9×, and 10–30×. Our work is released in the DataLife tool [44].

We conclude that DFLs are an effective methodology for gathering insight into actionable opportunities on workflows of contrasting styles. In our case studies, we explored remediation options ranging from the relatively easy to time-consuming, i.e., from changing coordination of tasks and data movement to exploring possible code changes with emulation. Although some of the remediation strategies appear straightforward in hindsight, *they were not apparent without the insight and guidance of DFLs and opportunity analysis*.

We believe such insight will be important as converged workflows become more common within experimental science. As workflows increasingly coordinate instrument-experiment control loops, a common challenge will be reworking the coordination between workflow components—originally designed for small-scale or throughput efficiency—into task and data movement rules that can deliver required response times and quality of service. Our future work includes exploring ways to automate suggestions for improved scheduling and resource assignment.

ACKNOWLEDGMENTS

We thank Ryan Friese and Burcu Mutlu (Pacific Northwest National Laboratory) for assistance with Belle II Monte Carlo results. We thank Stephen Young (Pacific Northwest National Laboratory) for alerting us to the terminology *caterpillar tree*.

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" and the "Cloud, HPC, and Edge for Science and Security" LDRD at Pacific Northwest National Laboratory. PNNL is operated by Battelle for the DOE under Contract DE-AC05-76RL01830.

REFERENCES

[1] 1000Genomes Workflow Git repo. https://github.com/pegasus-isi/1000genome-workflow. Accessed: 2023-03-15.
[2] Caterpillar tree. https://en.wikipedia.org/wiki/Caterpillar_tree. Accessed: 2023-03-15.
[3] DeepDriveMD Workflow Git repo. https://github.com/radical-collaboration/DeepDriveMD. Accessed: 2023-03-15.
[4] Montage Workflow Git repo. https://github.com/wfcommons/pegasus-instances/tree/master/montage. Accessed: 2023-03-15.
[5] Sankey Diagrams. https://www.data-to-viz.com/graph/sankey.html. Accessed: 2023-03-15.
[6] Seismic Cross Correlation Workflow Git repo. https://github.com/wfcommons/pegasus-instances/tree/master/seismology. Accessed: 2023-03-15.

[7] Hasan Abbasi, Matthew Wolf, Greg Eisenhauer, Scott Klasky, Karsten Schwan, and Fang Zheng. 2010. DataStager: scalable data staging services for petascale applications. *Cluster Computing* 13, 3 (2010), 277–290.

[8] Dong H. Ahn, Xiaohua Zhang, Jeffrey Mast, Stephen Herbein, Francesco Di Natale, Dan Kirshner, Sam Ade Jacobs, Ian Karlin, Daniel J. Milroy, Bronis De Supinski, Brian Van Essen, Jonathan Allen, and Felice C. Lightstone. 2022. Scalable Composition and Analysis Techniques for Massive Scientific Workflows. In *2022 IEEE 18th International Conference on e-Science (e-Science)*. 32–43.

[9] Michael Albrecht, Patrick Donnelly, Peter Bui, and Douglas Thain. 2012. Makeflow: A Portable Abstraction for Data Intensive Computing on Clusters, Clouds, and Grids. In *Proceedings of the 1st ACM SIGMOD Workshop on Scalable Workflow Execution Engines and Technologies* (Scottsdale, Arizona, USA) *(SWEET '12)*. Association for Computing Machinery, New York, NY, USA, Article 1, 13 pages.

[10] Xian-He Sun Anthony Kougkas, Hariharan Devarajan. 2020. I/O Acceleration via Multi-Tiered Data Buffering and Prefetching. *Journal of Computer Science and Technology* 35, 1, Article 92 (2020), 28 pages.

[11] U. Ayachit, A. Bauer, E. P. N. Duque, G. Eisenhauer, N. Ferrier, J. Gu, K. E. Jansen, B. Loring, Z. Lukic, S. Menon, D. Morozov, P. O'Leary, R. Ranjan, M. Rasquin, C. P. Stone, V. Vishwanath, G. H. Weber, B. Whitlock, M. Wolf, K. J. Wu, and E. W. Bethel. 2016. Performance Analysis, Design Considerations, and Applications of Extreme-Scale In Situ Infrastructures. In *Proc. of the Intl. Conf. for High Performance Computing, Networking, Storage and Analysis*. 921–932.

[12] Yadu Babuji, Anna Woodard, Zhuozhao Li, Daniel S. Katz, Ben Clifford, Rohan Kumar, Lukasz Lacinski, Ryan Chard, Justin M. Wozniak, Ian Foster, Michael Wilde, and Kyle Chard. 2019. Parsl: Pervasive Parallel Programming in Python. In *Proc. of the 28th Int. Symp. on High-Performance Parallel and Distributed Computing*. ACM, New York, NY, USA, 25–36.

[13] Martin Barisits, Thomas Beermann, Frank Berghaus, Brian Bockelman, Joaquin Bogado, David Cameron, Dimitrios Christidis, Diego Ciangottini, Gancho Dimitrov, Markus Elsing, , Markus Elsing, Vincent Garonne, Alessandro di Girolamo, Luc Goossens, Wen Guan, Jaroslav Guenther, Tomas Javurek, Dietmar Kuhn, Mario Lassnig, Fernando Lopez, Nicolo Magini, Angelos Molfetas, Armin Nairz, Farid Ould-Saada, Stefan Prenner, Cedric Serfon, Graeme Stewart, Eric Vaandering, Petya Vasileva, Ralph Vigne, and Tobias Wegner. 2019. Rucio: Scientific Data Management. *Computing and Software for Big Science* 3, 1 (Aug 2019).

[14] Nathan Beckmann, Haoxian Chen, and Asaf Cidon. 2018. LHD: Improving Cache Hit Rate by Maximizing Hit Density. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. USENIX Association, Renton, WA, 389–403.

[15] Alexander Brace, Shantenu Jha, Igor Yakushin, Hyungro Lee, Heng Ma, Anda Trifan, Li Tan, Todd Munson, Matteo Turilli, Ian Foster, and Arvind Ramanathan. 2022. Coupling streaming AI and HPC ensembles to achieve 100-1000× faster bio-molecular simulations. In *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE.

[16] Rodrigo N. Calheiros, Rajiv Ranjan, Anton Beloglazov, César A. F. De Rose, and Rajkumar Buyya. 2011. CloudSim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms. *Software: Practice and Experience* 41, 1 (2011), 23–50.

[17] Philip Carns, Kevin Harms, William Allcock, Charles Bacon, Samuel Lang, Robert Latham, and Robert Ross. 2011. Understanding and improving computational science storage access through continuous characterization. *ACM Transactions on Storage (TOS)* 7, 3 (2011), 1–26.

[18] Henri Casanova, Arnaud Giersch, Arnaud Legrand, Martin Quinson, and Frédéric Suter. 2014. Versatile, Scalable, and Accurate Simulation of Distributed Applications and Platforms. *J. Parallel and Distrib. Comput.* 74, 10 (June 2014), 2899–2917.

[19] Weiwei Chen and Ewa Deelman. 2011. Workflow Overhead Analysis and Optimizations. In *Proc. of the 6th Workshop on Workflows in Support of Large-Scale Science (WORKS '11)*. Association for Computing Machinery, New York, NY, USA, 11–20.

[20] Xuhao Chen and Arvind. 2022. Efficient and Scalable Graph Pattern Mining on *GPU*s. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. 857–877.

[21] Fahim Chowdhury, Francesco Di Natale, Adam Moody, Kathryn Mohror, and Weikuan Yu. 2022. DFMan: A Graph-based Optimization of Dataflow Scheduling on High-Performance Computing Systems. In *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 368–378.

[22] Laura Clarke, Xiangqun Zheng-Bradley, Richard Smith, Eugene Kulesha, Chunlin Xiao, Iliana Toneva, Brendan Vaughan, Don Preuss, Rasko Leinonen, Martin Shumway, et al. 2012. The 1000 Genomes Project: data management and community access. *Nature methods* (2012).

[23] Ewa Deelman, Karan Vahi, Gideon Juve, Mats Rynge, Scott Callaghan, Philip J. Maechling, Rajiv Mayani, Weiwei Chen, Rafael Ferreira da Silva, Miron Livny, and Kent Wenger. 2015. Pegasus, a workflow management system for science automation. *Future Gener Comput Syst* 46 (2015), 17–35.

[24] Hariharan Devarajan and Kathryn Mohror. 2022. Extracting and Characterizing I/O Behavior of HPC Workloads. In *Proc. of the 2022 IEEE Conf. on Cluster Computing*. IEEE.

[25] Sherif El-Basil. 1987. Applications of caterpillar trees in chemistry and physics. *Journal of mathematical chemistry* 1, 2 (1987), 153–174.

[26] Charles M Fiduccia and Robert M Mattheyses. 1988. A linear-time heuristic for improving network partitions. In *Papers on Twenty-five years of electronic design automation*. 241–247.

[27] Rosa Filgueira, Rafael Ferreira Da Silva, Amrey Krause, Ewa Deelman, and Malcolm Atkinson. 2016. Asterism: Pegasus and dispel4py hybrid workflows for data-intensive science. In *2016 Seventh International Workshop on Data-Intensive Computing in the Clouds (DataCloud)*. IEEE, 1–8.

[28] Ryan D. Friese, Burcu O. Mutlu, Nathan R. Tallent, Joshua Suetterlein, and Jan Strube. 2020. Effectively Using Remote I/O For Work Composition in Distributed Workflows. In *Proc. of the 2020 IEEE Intl. Conf. on Big Data*. IEEE Computer Society.

[29] M. R. Garey and David S. Johnson. 1979. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman.

[30] Derek Gaston, Chris Newman, Glen Hansen, and Damien Lebrun-Grandié. 2009. MOOSE: A parallel computational framework for coupled systems of nonlinear equations. *Nuclear Engineering and Design* 239, 10 (2009), 1768–1778.

[31] Binny S Gill and Dharmendra S Modha. 2005. SARC: Sequential Prefetching in Adaptive Replacement Cache.. In *USENIX Annual Technical Conference, General Track*. 293–308.

[32] Takanori Hara and Belle II computing group. 2015. Computing at the Belle II experiment. *J. Phys. Conf. Ser.* 664, 1 (2015).

[33] Frank Harary and Allen J Schwenk. 1973. The number of caterpillars. *Discrete Mathematics* 6, 4 (1973), 359–365.

[34] Bruce Hendrickson, Robert W Leland, et al. 1995. A Multi-Level Algorithm For Partitioning Graphs. *SC* 95, 28 (1995), 1–14.

[35] The Belle II Computing System. https://docs.belle2.org/record/628/.

[36] Julien Herrmann, M. Yusuf Özkaya, Bora Uçar, Kamer Kaya, and Ümit V. Çatalyürek. 2019. Multilevel Algorithms for Acyclic Partitioning of Directed Acyclic Graphs. *SIAM Journal on Scientific Computing* 41, 4 (2019), A2117–A2145.

[37] Xiameng Hu, Xiaolin Wang, Lan Zhou, Yingwei Luo, Chen Ding, and Zhenlin Wang. 2016. Kinetic Modeling of Data Eviction in Cache. In *Proc. of the 2016 USENIX Conference on Usenix Annual Technical Conference* (Denver, CO, USA) *(USENIX ATC '16)*. USENIX Association, Berkeley, CA, USA, 351–364.

[38] Florin Isaila, Javier Blas, Jesus Carretero, Robert Latham, and Robert Ross. 2010. Design and evaluation of multiple-level data staging for blue gene systems. *IEEE Transactions on Parallel and Distributed Systems* 22, 6 (2010), 946–959.

[39] Florin Isaila, Jesus Carretero, and Rob Ross. 2016. CLARISSE: A Middleware for Data-Staging Coordination and Control on Large-Scale HPC Platforms. In *2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*. 346–355.

[40] Alex BW Kennedy and H Riall Sankey. 1898. The thermal efficiency of steam engines. In *Minutes of the Proceedings of the Institution of Civil Engineers*, Vol. 134. Thomas Telford-ICE Virtual Library, 278–312.

[41] B. W. Kernighan and S. Lin. 1970. An efficient heuristic procedure for partitioning graphs. *The Bell System Technical Journal* 49, 2 (1970), 291–307.

[42] Sriram Lakshminarasimhan, David A. Boyuka, Saurabh V. Pendse, Xiaocheng Zou, John Jenkins, Venkatram Vishwanath, Michael E. Papka, and Nagiza F. Samatova. 2013. Scalable in Situ Scientific Data Encoding for Analytical Query Processing. In *Proc. of the 22nd Intl. Symp. on High-Performance Parallel and Distributed Computing*. Association for Computing Machinery, New York, NY, USA, 1–12.

[43] Matthew Larsen, James Ahrens, Utkarsh Ayachit, Eric Brugger, Hank Childs, Berk Geveci, and Cyrus Harrison. 2017. The ALPINE In Situ Infrastructure: Ascending from the Ashes of Strawman. In *Proce. of the In Situ Infrastructures on Enabling Extreme-Scale Analysis and Visualization*. Association for Computing Machinery, New York, NY, USA, 42–46.

[44] DataLife. https://github.com/pnnl/datalife.

[45] Hyungro Lee, Matteo Turilli, Shantenu Jha, Debsindhu Bhowmik, Heng Ma, and Arvind Ramanathan. 2019. Deepdrivemd: Deep-learning driven adaptive molecular simulations for protein folding. In *2019 IEEE/ACM Third Workshop on Deep Learning on Supercomputers (DLS)*. IEEE, 12–19.

[46] Qing Liu, Jeremy Logan, Yuan Tian, Hasan Abbasi, Norbert Podhorszki, Jong Youl Choi, Scott Klasky, Roselyne Tchoua, Jay Lofstead, Ron Oldfield, Manish Parashar, Nagiza Samatova, Karsten Schwan, Arie Shoshani, Matthew Wolf, Kesheng Wu, and Weikuan Yu. 2014. Hello ADIOS: the challenges and lessons of developing leadership class I/O frameworks. *Concurrency and Computation: Practice and Experience* 26, 7 (2014), 1453–1473.

[47] Glenn K Lockwood, Nicholas J Wright, Shane Snyder, Philip Carns, George Brown, and Kevin Harms. 2018. *TOKIO on ClusterStor: connecting standard tools to enable holistic I/O performance analysis*. Technical Report. Lawrence Berkeley National Lab.(LBNL), Berkeley, CA (United States).

[48] Kshitij Mehta, Ashley Cliff, Frédéric Suter, Angelica M. Walker, Matthew Wolf, Daniel Jacobson, and Scott Klasky. 2022. Running Ensemble Workflows at Extreme Scale: Lessons Learned and Path Forward. In *2022 IEEE 18th International Conference on e-Science (e-Science)*. 284–294.

[49] Andre Merzky, Matteo Turilli, Mikhail Titov, Aymen Al-Saadi, and Shantenu Jha. 2022. Design and Performance Characterization of RADICAL-Pilot on Leadership-Class Platforms. *IEEE Transactions on Parallel and Distributed Systems* 33, 4 (2022), 818–829.

[50] D. Oleynik, S. Panitkin, M. Turilli, A. Angius, S. Oral, K. De, A. Klimentov, J. C. Wells, and S. Jha. 2017. High-Throughput Computing on High-Performance Platforms: A Case Study. In *13th IEEE Intl. Conf. on e-Science*. 295–304.

[51] M. Yusuf Özkaya, Anne Benoit, and Ümit V. Çatalyürek. 2020. Improving Locality-Aware Scheduling with Acyclic Directed Graph Partitioning. In *Parallel Processing and Applied Mathematics*, Roman Wyrzykowski, Ewa Deelman, Jack Dongarra, and Konrad Karczewski (Eds.). Springer International Publishing, Cham, 211–223.

[52] Silvio Pardi, Guglielmo de Nardo, and Guido Russo. 2016. Computing at Belle II. *Nuclear and Particle Physics Proceedings* 273–275 (2016), 950–956.

[53] Tirthak Patel and Suren Byna. 2020. 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*.

[54] Kassian Plankensteiner, Radu Prodan, Matthias Janetschek, Thomas Fahringer, Johan Montagnat, David Rogers, Ian Harvey, Ian Taylor, Ákos Balaskó, and Péter Kacsuk. 2013. Fine-Grain Interoperability of Scientific Workflows in Distributed Computing Infrastructures. *Jrnl of Grid Computing* 11, 3 (01 Sep 2013), 429–455.

[55] Antoniu Pop and Albert Cohen. 2013. Openstream: Expressiveness and data-flow compilation of openmp streaming programs. *ACM Transactions on Architecture and Code Optimization (TACO)* (2013).

[56] Marko A Rodriguez and Peter Neubauer. 2010. Constructions from dots and lines. *Bulletin of the American Society for Information Science and Technology* 36, 6 (2010), 35–41.

[57] Mats Rynge, Gideon Juve, Jamie Kinney, John Good, Bruce Berriman, Ann Merrihew, and Ewa Deelman. 2013. Producing an infrared multiwavelength galactic plane atlas using montage, pegasus and amazon web services. In *23rd Annual Astronomical Data Analysis Software and Systems, ADASS, Conference*.

[58] Philipp Schaad, Tal Ben-Nun, and Torsten Hoefler. 2022. Boosting Performance Optimization with Interactive Data Movement Visualization. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis* (Dallas, Texas) *(SC '22)*. IEEE Press, Article 64, 16 pages.

[59] Malachi Schram. 2019. The data management of heterogeneous resources in Belle II. *EPJ Web Conf.* 214 (2019), 04031.

[60] S. R. Slattery, P. P. H. Wilson, and R. P. Pawlowski. 2013. *The Data Transfer Kit: A geometric rendezvous-based tool for multiphysics data transfer.* American Nuclear Society - ANS, United States.

[61] P. Subedi, P. Davis, S. Duan, S. Klasky, H. Kolla, and M. Parashar. 2018. Stacker: An Autonomic Data Movement Engine for Extreme-Scale Data Staging-Based In-Situ Workflows. In *Proc. of the Intl. Conf. for High Performance Computing, Networking, Storage, and Analysis*. 920–930.

[62] Joshua Suetterlein, Ryan D. Friese, Nathan R. Tallent, and Malachi Schram. 2019. TAZeR: Hiding the Cost of Remote I/O in Distributed Scientific Workflows. In *Proc. of the 2019 IEEE Intl. Conf. on Big Data*. IEEE Computer Society, 383–394.

[63] Qian Sun, Tong Jin, Melissa Romanus, Hoang Bui, Fan Zhang, Hongfeng Yu, Hemanth Kolla, Scott Klasky, Jacqueline Chen, and Manish Parashar. 2015. Adaptive Data Placement for Staging-Based Coupled Scientific Workflows. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Austin, Texas) *(SC '15)*. Association for Computing Machinery, New York, NY, USA, 12 pages.

[64] KyleM. Tarplee, Ryan Friese, AnthonyA. Maciejewski, and HowardJay Siegel. 2015. Efficient and Scalable Pareto Front Generation for Energy and Makespan in Heterogeneous Computing Systems. In *Recent Advances in Computational Optimization*. Studies in Computational Intelligence, Vol. 580. Springer International Publishing, 161–180.

[65] Matteo Turilli, Mark Santcroos, and Shantenu Jha. 2018. A comprehensive perspective on pilot-job systems. *ACM Computing Surveys (CSUR)* 51, 2 (2018), 1–32.

[66] Venkatram Vishwanath, Mark Hereld, Vitali Morozov, and Michael E. Papka. 2011. Topology-aware data movement and staging for I/O acceleration on Blue Gene/P supercomputing systems. In *SC '11: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–11.

[67] V. Vishwanath, M. Hereld, and M. E. Papka. 2011. Toward simulation-time data analysis and I/O acceleration on leadership-class systems. In *2011 IEEE Symp. on Large Data Analysis and Visualization*. 9–14.

[68] Carl A. Waldspurger, Nohhyun Park, Alexander Garthwaite, and Irfan Ahmad. 2015. Efficient MRC Construction with SHARDS. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*. USENIX Association, Santa Clara, CA, 95–110.

[69] Chen Wang, Jinghan Sun, Marc Snir, Kathryn Mohror, and Elsa Gonsiorowski. 2020. Recorder 2.0: Efficient parallel I/O tracing and analysis. In *2020 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 1–8.

[70] Teng Wang, Suren Byna, Bin Dong, and Houjun Tang. 2018. UniviStor: Integrated Hierarchical and Distributed Storage for HPC. In *2018 IEEE International Conference on Cluster Computing (CLUSTER)*. 134–144.

[71] Teng Wang, Shane Snyder, Glenn Lockwood, Philip Carns, Nicholas Wright, and Suren Byna. 2018. Iominer: Large-scale analytics framework for gaining knowledge from i/o logs. In *2018 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 466–476.

[72] Michael Wilde, Mihael Hategan, Justin M. Wozniak, Ben Clifford, Daniel S. Katz, and Ian Foster. 2011. Swift: A language for distributed parallel scripting. *Parallel Comput.* 37, 9 (2011), 633 – 652.

[73] Jake Wires, Stephen Ingram, Zachary Drudi, Nicholas J. A. Harvey, and Andrew Warfield. 2014. Characterizing Storage Workloads with Counter Stacks. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. USENIX Association, Broomfield, CO, 335–349.

[74] Kanit Wongsuphasawat, Daniel Smilkov, James Wexler, Jimbo Wilson, Dandelion Mane, Doug Fritz, Dilip Krishnan, Fernanda B Viégas, and Martin Wattenberg. 2017. Visualizing dataflow graphs of deep learning models in tensorflow. *IEEE transactions on visualization and computer graphics* (2017).

[75] Junyao Yang, Yuchen Wang, and Zhenlin Wang. 2021. Efficient Modeling of Random Sampling-Based LRU. In *50th International Conference on Parallel Processing* (Lemont, IL, USA) *(ICPP 2021)*. Association for Computing Machinery, New York, NY, USA, Article 32, 11 pages.

[76] Yu Zhang, Ping Huang, Ke Zhou, Hua Wang, Jianying Hu, Yongguang Ji, and Bin Cheng. 2020. OSCA: An Online-Model Based Cache Allocation Scheme in Cloud Block Storage Systems. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, 785–798.

[77] Weiming Zhao, Xinxin Jin, Zhenlin Wang, Xiaolin Wang, Yingwei Luo, and Xiaoming Li. 2011. Low Cost Working Set Size Tracking. In *Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference* (Portland, OR) *(USENIXATC'11)*. USENIX Association, USA, 17.

[78] Tong Zhou, Jun Shirako, Anirudh Jain, Sriseshan Srikanth, Thomas M. Conte, Richard Vuduc, and Vivek Sarkar. 2020. Intrepydd: Performance, Productivity, and Portability for Data Science Application Kernels. In *Proceedings of the 2020 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software* (Virtual, USA) *(Onward! 2020)*. Association for Computing Machinery, New York, NY, USA, 65–83.

Appendix: Artifact Description/Artifact Evaluation

ARTIFACT IDENTIFICATION

A critical performance challenge in distributed scientific workflows is coordinating tasks and data flows on distributed resources. To guide these decisions, this paper introduces data lifecycle analysis. Workflows are commonly represented using directed acyclic graphs (DAGs). Data lifecycles enrich task DAGs with data objects and properties that describe data flow and how tasks interact with that flow. Lifecycles enable analysis from several important perspectives: task, data, and data flow. This paper describes representation, measurement, analysis, visualization, and opportunity identification for data lifecycles. Our measurement is both distributed and scalable, using space that is constant per data file. We use lifecycles and opportunity analysis to reason about improved task placement and reduced data movement for five scientific workflows with different characteristics. Case studies show improvements of 15x 1.9x, and 10–30x.

Our work is implemented in the DataLife toolsuite. There are three steps.

The first step is monitoring a workflow's I/O during executions with representative inputs. DataLife's monitoring tool overrides I/O calls using typical dynamic linking techniques e.g., Linux LD_PRELOAD. DataLife's focuses on POSIX and C I/O, which includes all variants of open, close, read, write, fseek, etc.

Second, DataLife contructs a data lifecycle graph (DLG). A DLG is a property graph, where both vertices and edges can be annotated with property values formed from raw histogram statistics. Both tasks and data files become vertices in the graph. A read operation (by a consumer task) is denoted by a directed edge from a data source (vertex) to a task sink. A write operation (by a producer task) is denoted by a directed edge from a task source to data sink. Each directed edge in the graph is annotated with a lifecycle property value representing execution times, data volumes, footprints, flow rates, and access patterns.

The third step is analysis and visualization. To visualize data lifecycle graphs, DataLife generates Sankey diagrams, a representation designed for data flows. To quickly highlight interesting graph entities and relations, DataLife associates property values with them, forms a projection of the entity, and rank orders the results. Finally, DataLife finds opportunities for performance improvement based on DLG patterns.

REPRODUCIBILITY OF EXPERIMENTS

We have focused on four scientific workflows in the paper. See the description below.

1000Genome: The 1000 Genome project studies the genomes of 2504 individuals from 26 populations to understand human genetic variation. The source code of 1000Genome is available at https://github.com/pegasus-isi/1000genome-workflow.

DeepDriveMD: DeepDriveMD is a deep learning-driven molecular dynamics simulations workflow for protein folding. Its source code is available at https://github.com/DeepDriveMD/DeepDriveMD-pipeline.

Montage: Montage is a compute-intensive astronomical image processing toolkit that uses a collection of structural images to build a composite image mosaic for viewing outer space. Its source code is available at https://github.com/pegasus-isi/montage-workflow-v3.

Seismic: Seismic Cross Correlation is a data-intensive workflow that computes cross correlation of signals from multiple seismic stations. Its source code is avaiable at https://github.com/pegasus-isi/seismology-workflow.

The code artifact includes three parts: the monitoring tool, the program and scripts for visualizing the DLGs, and the performance evaluation scripts for analysis. Please find our code for visualization and evaluation at https://gitlab.pnnl.gov/perflab/bigflowtools/datalife/-/tree/main/evaluation_scripts/. The monitoring tool is located at https://github.com/pnnl/tazer, under the 'feature_trackfile' branch. The visualization part is located under the DLG folder, in which you can find the Python notebook for producing the DLG graph for the scientific workflows 1000Genome, DeepDriveMD, Montage, and Seismic. The visualization takes the collected I/O states as input and generates the DLG graph (Figure 3), ranking tables (Figure 3), and caterpillar trees (Figure 4 and Figure 5 for 1000Genome only) for each scientific workflow. For convenience, we have generated the I/O state and put it in the 'tazer_stat' under each workflow's DLG directory.

The performance evaluation part is located under the 'Performance' folder, in which you can find the evaluation code for 1000Genome and DeepDriveMD. The Python notebooks will generate the performance plots for 1000Genome and DeepDriveMD in Figures 6 and 7. These Python notebooks take as input the performance numbers generated by the source code under '*
$_{p}erf$
$_{n}umber$'$directories.Togeneratetheperformance, lookfortheSBATCHscriptswith$

$_{p}erf$
$_{n}umber$'$directoriesareeitherinputorbinaryfiles.Forconvenience, wehavesavedth$

$_{b}in$'$underthe$' *
$_{p}erf$
$_{n}umber$'$directories.PleaseseetheREADMEsundereachdirectoryforfurtherguida$

Monitoring workflow execution with DataLife/collector:

The DataLife/collector generates a database with relevant statistics for each task and data file used by the workflow. Currently, each task-data pair is a separate file. The time for this step is equivalent to the time of executing the workflow, as the overhead of monitoring lifetimes of storage accesses is negligible.

The monitoring library can be obtained and compiled by following these steps:

- git clone https://github.com/pnnl/tazer && cd tazer - git checkout -b feature_trackfile origin/feature_trackfile - mkdir build && cd build - cmake -DCMAKE_INSTALL_PREFIX=$INSTALL_DIR .. - make VERBOSE=1 && make install

In the following data collection stage we assume that the $INSTALL_DIR is set to $DATALIFE_PATH/datalife/build.

To launch the DataLife/collector, we simply adjust the launch of workflow tasks to preload the monitoring library. We give an example for the DDMD and 1000 Genomes workflows. (The example omits parallelism.) Other workflows follow exactly the same pattern.

DDMD:

LD_PRELOAD=$DATALIFE_PATH/datalife/build/src/client/libclient.so python $PATH_to_deepdrivemd/sim/openmm/run_openmm.py -c md_stage.yaml

LD_PRELOAD=$DATALIFE_PATH/datalife/build/src/client/libclient.so python $PATH_to_deepdrivemd/aggregation/basic/aggregate.py -c agg_stage.yaml

LD_PRELOAD=$DATALIFE_PATH/datalife/build/src/client/libclient.so python $PATH_to_deepdrivemd/models/aae/train.py -c machine_learning_stage.yaml

LD_PRELOAD=$DATALIFE_PATH/datalife/build/src/client/libclient.so python $PATH_to_deepdrivemd/agents/lof/lof.py -c inference_stage.yaml

1000 Genomes:

LD_PRELOAD=$DATALIFE_PATH/datalife/build/src/client/libclient.so $PATH_to_1000genome-workflow/bin/individuals.py ALL.chr1.250000.vcf 1 1 2 30

LD_PRELOAD=$DATALIFE_PATH/datalife/build/src/client/libclient.so $PATH_to_1000genome-workflow/bin/individuals.py ALL.chr1.250000.vcf 1 2 3 30

…

LD_PRELOAD=$DATALIFE_PATH/datalife/build/src/client/libclient.so $PATH_to_1000genome-workflow/bin/individuals.py ALL.chr1.250000.vcf 1 29 30 30

LD_PRELOAD=$DATALIFE_PATH/datalife/build/src/client/libclient.so $PATH_to_1000genome-workflow/bin/individuals_merge.py 1 chr1n-1-2.tar.gz chr1n-2-3.tar.gz … chr1n-29-30.tar.gz

LD_PRELOAD=$DATALIFE_PATH/datalife/build/src/client/libclient.so $PATH_to_1000genome-workflow/bin/sifting.py ALL.chr1.phase3_shapeit2_mvncall_integrated_v5.20130502.sites.annotation.vcf 1

LD_PRELOAD=$DATALIFE_PATH/datalife/build/src/client/libclient.so $PATH_to_1000genome-workflow/bin/mutation_overlap.py -c 1 -pop SAS

LD_PRELOAD=$DATALIFE_PATH/datalife/build/src/client/libclient.so $PATH_to_1000genome-workflow/bin/frequency.py -c 1 -pop SAS

Data Lifecycle analysis with DataLife/analyzer:

The following overviews analysis with DataLife/analyzer.

1) DataLife/analyzer is implemented in Python and leverages the libraries NetworkX and Plotly. The setup configuration files (requirements.txt for python virtualenv packages and environment.yml for conda) will be provided to prepare necessary python environments.

2) DLG analysis begins by reading lifecycle measurements using 'df = read_tazer_stats()'.

3) We use Python NetworkX to store the workflow structure in a Data Lifecycle Directed Acyclic Graph (DL-DAG) with lifecycle and flow statistics: 'G, pos = get_graph(wf, df)'. In the DL-DAG, we distinguish two types of nodes, 'task' and 'data', using the 'ntype' node attribute. For edges, the 'value' attribute is used to store the metric values of the task-data relations.

4) To visualize data lifecycles, we use the Python Plotly library to generate Sankey Diagrams. Specifically, we use two functions, 'get_nodes_for_sankey()' and 'get_links_for_sankey()', which return a Python dictionary for Sankey from a graph object. We use colors to highlight the different node types (task node-'red', data node-'blue'), sing a color map ({'task': 'red', 'file': 'blue' }) and the width of the edges reflects the 'value' attribute with a gray color by default. We update the corresponding edge pairs with purple color for the critical path, according to different properties, using the 'update_edge_cpath()' function.

5) We prune the graph to a caterpillar tree by removing nodes bigger than 1 degree from the critical path via the 'draw_caterpillar()'.

6) We continue using a graph object to obtain a ranking table provided by Pandas DataFrame, using the 'table_for_producer_consumer()'.

Example of Data Lifecycle Analysis Steps:

The following steps provide an example to reproduce experiments reported in the paper using the GitLab repository: https://gitlab.pnnl.gov/perf-lab/bigflowtools/datalife/-/tree/main/evaluation_scripts/.

1) To generate the DataLifecycle graphs, ranking table, and caterpillar tree in Figures 3, 4 and 5, please go to the DLG folder.

2) To generate the execution time plots in Figures 6, 7 and 8, please go to the Performance folder.

3) For the 1000 Genome workflow, find the DLG/1000genome_DLG folder and the Python notebook for producing the ranking table, Data Lifecycle Graph (Figure 3a), and Caterpillar tree (Figure 4a and Figure 5)

4) For the performance plot for 1000 Genome (Figure 6), find the performance/1000genome_plot folder and the performance numbers in Cell #3 need to manually update from results generated from the SBATCH scripts located under 1000genome_perf_number.

Estimation of the execution time to execute the experiment workflow:

Our DLG analysis is quick and efficient and can execute each experiment workflow in less than a few minutes. We observe the artifacts to complete our DLG results in a minute or so from the experiments in the paper. It may execute larger experiments in a relatively short amount of time.