FastFlow: Rapid Workflow Response By Prioritizing Critical Data Flows and their Interactions

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

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

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

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

Nathan R. Tallent
Pacific Northwest National Laboratory
Future Computing Technologies
Richland, WA, USA
tallent@pnnl.gov

Zhen Peng
Pacific Northwest National Laboratory
Richland, CA, USA
zhen.peng@pnnl.gov

Abstract

Scientists are automating instrument operation using distributed scientific workflows. To improve workflow response time, we present a new scheduling method called FastFlow. Although there has been much prior work on scheduling, we introduce a new insight: prioritizing critical flow paths and their *interactions*—in/out flow paths—enables a linear-time scheduling method that achieves high quality results. Since many workflows are executed repeatedly, our method is based on a monitor-analyze-optimize strategy. After monitoring a workflow's execution-time *data flow*, our method identifies response-critical paths and their in/out data flows using a linear-time partitioning algorithm. For each partition, a greedy linear-time scheduler selects between the better of flow parallelism and flow locality. The resulting schedules are high quality because the greedy decisions avoid delaying or shifting the critical flow. We evaluate a range of representative workflows and compare against state-of-the-art methods. Our experiments demonstrate mean speedups of 1.15×, 3.5×, 1.04×, and 1.07× compared to the next best, which are *not* linear time. Compared to popular linear-time methods, speedups are up to 1.28×, 87×, 1.4×, and 5×.

CCS Concepts

• **General and reference** → **Performance**; *Measurement*; • **Computing methodologies** → **Distributed computing methodologies**; • **Information systems** → **Distributed storage**; *Hierarchical storage management*.

Keywords

distributed workflows, data flow lifecycles, graph partitioning, resource assignment, storage bottlenecks

SSDBM 2025, Columbus, OH, USA

ACM ISBN 979-8-4007-1462-7/25/06
https://doi.org/10.1145/3733723.3733735

ACM Reference Format:
Jesun Sahariar Firoz, Hyungro Lee, Luanzheng Guo, Meng Tang, Nathan R. Tallent, and Zhen Peng. 2025. FastFlow: Rapid Workflow Response By Prioritizing Critical Data Flows and their Interactions. In *The International Conference on Scalable Scientific Data Management 2025 (SSDBM 2025), June 23–25, 2025, Columbus, OH, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3733723.3733735

1 Introduction

To increase the pace of scientific discovery, scientists are automating instrument operation using control loops. These loops replace most human decisions with a workflow composed of sub-applications — AI/ML, data analytics, and numerical solvers — and large amounts of data movement [13, 18, 22, 24, 43–45]. The faster these loops, the faster the pace of experiments. The key bottleneck is rarely with instruments but with the computational loop and its use of compute units, data flow, or storage. Removing bottlenecks often requires improved coordination and resource assignment, i.e., the combination of determining (a) *allocation*, or what resources to allocate, (b) *scheduling*, or when tasks and data flow execute, and (c) *assignment*, or the resources on which data flows and tasks execute.

Workflow managers and schedulers represent workflow execution using directed acyclic graphs (DAGs) of control and data dependencies (cf. Figure 1). Prior work on scheduling and resource assignment has prioritized different things: flow between DAG clusters [23, 27, 29, 32–34, 46], critical paths [6, 14, 39, 42, 49, 50], or data (locality, reuse, etc.) [19, 21, 28, 35, 47].

This paper introduces a new method of workflow scheduling and resource assignment, named FastFlow, based on the following insight: *for each independent critical flow path, use a fast (locally greedy) scheduler that uses the critical flow's interactions—in/out flow paths—to select between the better of flow parallelism and flow locality.* Figure 1 compares the results of FastFlow with two other state-of-the-art methods, DFMan [16] and FaaSFlow [39]. Subfigure (a) highlights a critical flow (green) and the interactions—in/out flow paths (orange and blue)—that FastFlow prioritizes for scheduling. These interactions should also be prioritized—assigned to sufficiently fast storage—*so that critical flows are neither delayed nor shifted.* The challenge is doing this in linear time.

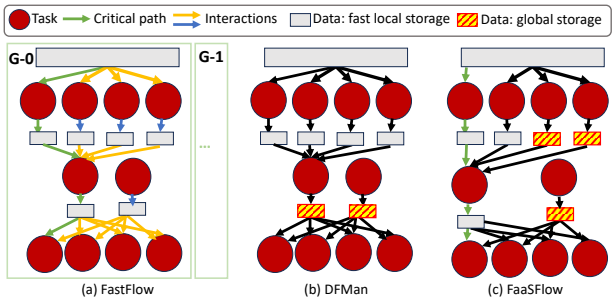


Figure 1: Comparison of FastFlow with state-of-the-art DFMan [16] and FaaSFlow [39]. Each approach schedules a directed acyclic graph (DAG) where squares and circles denote data and task vertices, respectively. (a) FastFlow assigns all data to fast (local) storage (gray) in linear time. It reasons about *flow interactions*—in/out flow paths (orange and blue edges)—that may *delay* or *shift* the critical path. Independent interaction sets form partitions (G-0 and G-1). The partitions provide the "right" scheduling context for a local greedy scheduler that selects the better of flow parallelism vs. data locality, all while ensuring co-location of data flows and tasks. (b) DFMan may place some data on slower global storage (yellow/red), delaying the critical flow. DFMan prioritizes consumer edges over interacting flows; and optimizes overall *data bandwidth* rather than response time. (c) FaaSFlow prioritizes tasks on the critical path but may introduce delays (yellow/red) by scheduling tasks from the interacting flows later. Finally, FastFlow's partitions highlight flow parallelism that neither DFMan nor FaaSFlow recognize.

Since many workflows are executed repeatedly, we use a monitor-analyze-optimize strategy. The *monitoring* step (Section 2.1) obtains a data flow lifecycle graph (DFL), a DAG that highlights dynamic data flow between tasks and is annotated with detailed flow statistics. The *analysis* partitions the graph (Section 3; G-*x* in Figure 1(a)) by each independent critical flow path. Each partition contains a critical flow (green edges) and the paths that may either *delay* the flow or *shift* its criticality (orange and blue). We say that criticality *delays* come from *incoming* paths; *shifts* from *outgoing* paths. Finally, each partition is *optimized* (Section 4) with a greedy scheduler that further divides each partition into sets of producer-consumer subpaths that are best assigned to the same compute, storage, and flow resources. For each subpath set, the scheduler selects between the better of flow parallelism and flow locality, guided by tradeoff analysis (Section 5). The partitions (step 2) ensure that each set of subpaths (step 3) are scheduled on the fastest possible local storage, resulting in near-optimal data flows that cause the Figure's speedup on a difficult but important workflow. Unlike DFMan and FaaSFlow, FastFlow's partitioning and scheduling algorithm is linear in DFL edges (given fixed resources). FastFlow will be open-sourced.

We evaluate (Section 6) FastFlow on representative workflows and compare against workflow scheduling methods to optimize critical paths (FaaSFlow [39], CPOP [49], and HEFT), data flows (METIS [32, 33] and dagP [29]), and storage bandwidth (DFMan [16]). Our experiments demonstrate geo-mean speedups of 1.15×, 3.5×, 1.04×, and 1.07× compared to next best (*not* linear time); and speedups

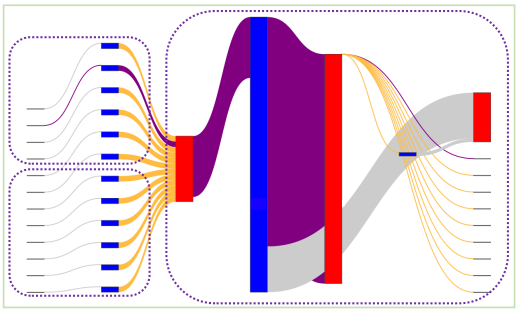


Figure 2: A data flow lifecycle (DFL) graph, visualized as a Sankey diagram. Flow edges (directed, left-to-right) connect vertices representing tasks (red) and data (blue). This DFL's edges show flow volume (thickness). Purple edges show a critical path. This DFL shows one partition (green rectangle) with three segments (dotted purple).

of up to 1.28×, 87×, 1.4×, and 5× compared to popular linear-time methods. FastFlow is linear time. Our contributions are:

- Fast method to improve a workflow's *resource allocation, schedule, and resource assignment* by focusing on critical data flow paths and their in/out interaction flows.
- Two-level workflow graph partitioning (linear in edges) creating (a) path groups and (b) sets of subpaths that enable reasoning about the impact of critical flows and their dependent producer-consumers.
- Scheduling that selects between the better of biasing flow parallelism or flow locality. The tradeoff analysis accounts for I/O operation type, pattern, and size.
- Evaluation on workflows with different characteristics and comparison against state-of-the-art methods.

2 Overview

Our method, called FastFlow, introduces a fast (linear-time) method to improve data flow along both entire flow paths and their individual producer-consumers. Since many workflows are executed repeatedly, it adopts a monitor-analyze-optimize strategy where monitoring and re-scheduling can be amortized over repeated executions.

2.1 Characterizing critical data flows

FastFlow leverages a monitoring tool that obtains a graph representation, called a *data flow lifecycle graph* [35], that highlights dynamic data flow between tasks. Workflows are commonly represented using directed acyclic graphs (DAGs) of tasks. A *data flow lifecycle graph* [35] enriches task DAGs with data objects and properties that describe data flow and how tasks depend on that flow. Figure 2 shows an example; flow proceeds from left to right. Within the DFL graph, data flows are composed of producer-consumer relations that can differ by inflow and outflow parallelism. Data flow corresponds to an edge: a *producer* is an edge from task to data vertex; a *consumer* is the reverse. A producer-consumer flow (two edges) is semantically distinct from I/O operations because it represents generation and consumption of the *same* datum. All

graph components, including data vertices and flow edges, are annotated with property values representing execution statistics such as execution time, data access type, size, rates, reuse, etc. These annotations guide our tradeoff analysis and scheduling.

2.2 Guidance from critical flows and their interactions

Our analysis then partitions the DFL graph. Consider Figure 1. Each partition (G-x) consists of an independent critical flow path (green edges) and its interacting paths (orange and blue), i.e., the in/out paths that may either *delay* or *shift* its criticality. The interactions should be prioritized during scheduling so that critical flows are not delayed (yellow/red boxes). Within a partition, in/out interactions identify data and flow locality that can be exploited when assigned to fast storage, resulting in fast data flow. Areas with few interactions between them can be scheduled for parallelism.

2.3 Scheduling and tradeoff analysis

Scheduling then assigns tasks and data to resources based on the most beneficial of two strategies: maximize flow parallelism or flow locality. In the process, it further sub-partitions each partition into contiguous *segments*. Figure 2 shows partitions and segments. The two-level partitioning informs scheduling by prioritizing the tasks and data most likely to impact the critical flows. Partitions and segments enable reasoning about flow parallelism and locality, each flow's dependencies, and the producer-consumer relations that are most likely to impact response time. Thus, FastFlow's schedule in Figure 1 selects the fastest storage for each segment, unlike the alternatives. Additionally, FastFlow's partitioning and scheduling are linear in graph edges, compared to the alternative approaches.

2.4 Modeling flows

To select between scheduling strategies (parallelism vs. locality) and to determine the best flow resource (storage assignment for data), it is necessary to project the performance of concurrent flows within a segment. The projection should account not only for task time, but also flow: data generation and consumption, including its distinct operations and patterns. In our case, data flow occurs within file systems, whether implicitly (shared) or explicitly (copies).

Our methodology leverages a one-time preparatory step that creates a resource model for each workflow pattern and storage resource. To model a *storage resource*, we employ ERT4IO [52] and request models for distinct operation types (read vs. write), concurrency levels, and sizes. To model a *workflow segment's flow*, we use the DFL's flow statistics for each producer and consumer, such as expected concurrency, operation type, pattern, and operation size.

3 Caterpillar Partitioning

To form the right context for a local scheduler, we partition the DFL. To partition, we find the *critical path*, the longest path in a directed acyclic graph that dictates final workflow response time. Next, we find the path's nearby interactions by forming the path's corresponding *caterpillar tree* [2, 20, 26, 35] that includes (a) for a data node, the (distance 1) consumer task; (b) for a task, the (distance 1) vertices from (to) its incoming (and outgoing) data; and the (distance 2) paths to subsequent (and dependent) tasks if

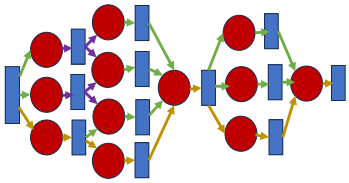


Figure 3: Comparing DFL critical path (gold edges), caterpillar tree (gold and green) and grove (gold, green and violet edges). Vertices along the edges are included in the corresponding graph structure. Blue rectangles denote data vertices; red circles task vertices.

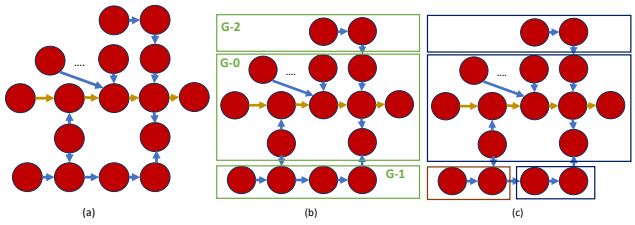


Figure 4: Partitioning a DFL graph via caterpillar DAGs. Critical path marked with gold edges. (a) A DFL graph (tasks only); (b) corresponding partitions (green rectangles; `G-0`, `G-1`, `G-2`); and c) acyclic partitions (black rectangles).

there is no join with the critical path. Finally, we extend the nearby interactions into in/out paths with the *caterpillar DAG*. Intuitively, a grove is the induction set of caterpillar trees implied by critical path. Formally, a *caterpillar DAG* is a vertex-induced subgraph, *rooted* at the starting vertex along the critical path, and the induction set consists of all the (data and task) vertices reachable from the root, as well as the task and data vertices on which any task vertex in the reachability set depends on. Figure 3 compares a DFL graph's critical path, caterpillar *tree* and caterpillar *grove*.

Given a DFL graph $\mathcal{G}$, partitioning decomposes it into a set $\mathcal{P}$ of caterpillar DAGs. Our algorithm for partitioning is shown in Algorithm 1. Figure 4(b) shows an example of caterpillar partitions. Figure 4(c) compares ours with common acyclic hierarchical methods [29, 46]. The acyclic partitioning "cuts" the critical flow, something FastFlow's partitions never do. Thus, acyclic partitioning prioritizes scheduling by partition rather than critical flow.

The algorithm starts by identifying the most critical path in $\mathcal{G}$, shown in gold in Figure 4(b). To complete the partition, it constructs the caterpillar DAG C along that path, or `G-0`. Next, it extracts C, removing all of its vertices from $\mathcal{G}$. Partitioning then proceeds to the residual's critical path and repeats until the residual is empty. This step identifies `G-1` as the next most important caterpillar DAG in the residual graph. Finally, it completes with `G-2`.

The caterpillar partitions identify potential flow parallelism that can be used to guide resource allocation/assignment (Section 4). Specifically, Lines 5–10 detect whether the current caterpillar DAG can be executed independently in parallel with other independent caterpillar DAGs. Workflows frequently employ similar sets of stages to concurrently process different datasets (e.g., chromosomes in 1000 Genomes, color bands in Montage).

Algorithm 1: Decomposing a DFL into caterpillar DAGs

Input: A DFL graph, $\mathcal{G}(\mathcal{V}, \mathcal{E})$
Output: Caterpillar DAGs, $\mathcal{P}$, ordered by their priorities

$\mathcal{G}_g(\mathcal{V}_g, \mathcal{E}_g) \leftarrow \mathcal{G}(\mathcal{V}, \mathcal{E})$, $\mathcal{P} \leftarrow \emptyset$ // priority queue for groves
while $\mathcal{V}_g \neq \emptyset$ **do**
  $\mathcal{W} \leftarrow$ FindCritPath($\mathcal{G}_g$), $C \leftarrow$ FindCaterpillarGrove($\mathcal{W}$)
  $\mathcal{V}', \mathcal{E}' \leftarrow$ vertices and edges on C
  foreach $v \in \mathcal{V}'$ **do** // Find independent caterpillar DAG
    $E_+(v) \leftarrow \bigcup e(u, v) \in \mathcal{E}', \forall u$ // incoming edges
    $E_-(v) \leftarrow \bigcup e(v, w) \in \mathcal{E}', \forall w$ // outgoing edges
    foreach $e(i, v) \in E_+(v)$ **do** // Same for outgoing edges
      if $i \notin \mathcal{V}'$ **then** *independent* $\leftarrow$ *false*, **break**
  if *independent* **then** C.mark_independent $\leftarrow$ *true*
  $\mathcal{P}$.insert(C), $\mathcal{V}_g \leftarrow \mathcal{V}_g \setminus \mathcal{V}'$, $\mathcal{E}_g \leftarrow \mathcal{E}_g \setminus \mathcal{E}'$
return $\mathcal{P}$

The algorithm also ranks partitions for scheduling priority (Section 4). Each partition's priority is decided by a compound set of metrics, including path length, maximum transferred data volume, data reuse, and approximate flow rate. With other methods, many near critical paths (FastFlow partitions), such as both of those in Figure 4, are given equal priority.

4 Scheduling and Resource Assignment

After caterpillar partitions $\mathcal{P}$ are prioritized, scheduling sub-partitions each one into *segments* and assigns the segments to resources. This two-level partitioning constrains the scheduling and assignment problem and enables scheduling for the best of either flow parallelism or data locality. The decisions are guided by tradeoff analysis and models. The tradeoff analysis ensures each segment can execute within the resource's storage capacity constraints. The objective is to improve overall response time.

Algorithm 2 shows the steps. The algorithm's inputs are the DFL partitions $\mathcal{P}$ and a resource availability matrix M. In the tradeoff analysis, the DFL is used to characterize specific flows (Algorithm 2 lines 6, 8). The matrix is used to characterize available resources and project performance of proposed flows. The matrix is generated during modeling (Section 5).

The resource availability matrix has dimensions $R \times N$, where each row represents available resources and columns represent an n-tuple describing each resource. The resources consist of a combination of available compute node types and flow resources. Specifically, the R resource tuples { *compute* × *storage* × *I/O pattern* } represent computational and flow performance in terms of compute cores, storage type and capacity, and flow performance models that generate values in operations/s and bytes/s. Given a producer or consumer flow, evaluation of the resource matrix makes a performance decision for each resource.

The algorithm is based on the following realistic assumptions. First, the algorithm cannot change task synchronization primitives, such as would occur with fine-grained pipelining (e.g., converting whole-file synchronization into push-pull sub-file transfers). Second, the DFL graph is sufficiently representative of the execution. We can ensure this by collecting DFLs on relevant workflow inputs.

Algorithm 2: Partitioning caterpillar DAGs into caterpillar segments and assigning resources to them

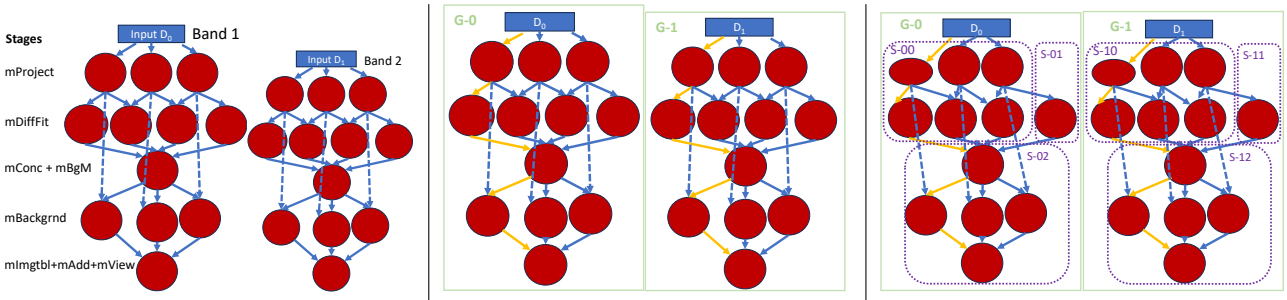
Input: A DFL graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$, annotated with flow statistics, caterpillar DAGs, $\mathcal{P}$, a resource availability matrix $M_{R \times N}$ for each caterpillar DAG and r resource combinations.
Output: Caterpillar segments with assigned resources, $\mathcal{S}$

1 $\mathcal{S} \leftarrow \emptyset$, $\mathcal{D} \leftarrow$ CreateMap(), $l \leftarrow 0$ // l: level
2 **foreach** $\mathcal{F}_i \in \mathcal{P}$ **do**
3 $\mathcal{V}', \mathcal{E}' \leftarrow$ vertices and edges on $\mathcal{F}_i$
4 $\mathcal{PC} \leftarrow \{(\mathcal{P}_j, C_j) | (p_{jk}, c_{jk'}) \in \mathcal{E}', p_{jk} \in P_j, c_{jk'} \in C_j, \forall$ producer-consumer (P-C) relation j and task $k \in \mathcal{F}_i\}$
5 $\mathcal{V}_l \leftarrow \{v_i | v_i \in \mathcal{V}', \text{in-degree}(v_i) = 0\}$ // Start from root
6 $\{\mathcal{P}_r, C_r\}_l \leftarrow \mathcal{PC}(\mathcal{V}_l)$ // get all PC relations in $\mathcal{V}_l$
7 **while** $\mathcal{PC} \neq \emptyset$ **do**
8 $pat \leftarrow \{f_\#, d_{sz}, f_{bw}\} \leftarrow$ FlowNum_DataSz_FlowBW($\{\mathcal{P}_r, C_r\}_l$)
9 $r : \{core_\#, storage_{sz}, bw\} \leftarrow$ FindBestResource(M, pat)
10 **if** $f_\# > core_\#$ ***or*** $d_{sz} > storage_{sz}$ ***or*** $f_{bw} > bw$ **then**
11 $t \leftarrow \min\{f_\#/core_\#, d_{sz}/storage_{sz}, f_{bw}/bw\}$
12 $taskgroup \leftarrow \{g_d, d = 0, 1, \ldots t\}$, where $g_d \leftarrow \{(p_{jk}, c_{jk'}), \forall k \in 0, 1, \ldots,$ groupsize$\}$
13 $\mathcal{D} \leftarrow$ UpdateDataReuseInfo($\mathcal{D}$)
14 $strategy \leftarrow$ AnalyzeTradeoffs(r, $taskgroup$, $\mathcal{D}$)
15 **if** $strategy ==$ *maximize_flow_locality* **then**
16 $\mathcal{S}$.append(GrowSegHorizontallyAndAssign_local())
17 **else**
18 $\mathcal{S}$.append(GrowSegHorizontallyAndAssign_dist())
19 **else**
20 GrowSegVerticallyAndAssign()
21 **if** $\mathcal{PC} == \emptyset$ **then** $\mathcal{S}$.append(current-segment)
22 Adjust(l), $\mathcal{PC} \leftarrow \mathcal{PC} \setminus \{\mathcal{P}_r, C_r\}_l$
23 $\mathcal{V}_l \leftarrow \{v_i | v_i \in \mathcal{V}', v_i \in$ neighbor-list$(C_r)\}$, $\{\mathcal{P}_r, C_r\}_l \leftarrow \mathcal{PC}(\mathcal{V}_l)$
24 **return** $\mathcal{S}$

Note that the DFL can be dynamic as long as it is constrained. Finally, there is a fixed and sufficient (i.e., minimum) pool of resources for the duration of execution.

Partitioning caterpillar DAGs into segments. A critical step is partitioning caterpillar DAGs into *segments*. A *caterpillar segment* is a grouping of several sub-paths, each of consecutive tasks, whose subflows may benefit from increased data locality (vs. increased parallelism). For each caterpillar DAG $\mathcal{F}_i$, the algorithm traverses the DAG's vertices in a top-down, level-by-level fashion (breadth-first search order), starting from the root. On each level, the algorithm decides whether to grow or close a segment. Segments can grow *vertically* or *horizontally*. Assuming that paths proceed top-down, vertical segmentation is dictated by the number of consecutive tasks (spanning multiple levels) that benefit from co-location. Horizontal segmentation is dictated by the best number of *paths* that can be packed on a single compute resource without exceeding the local storage capacity and flow rate constraints.

At level l, the algorithm considers the next one or two consecutive levels $l+1$ and $l+2$, depending on the vertex type (task or data). To make a decision at level l for task vertex types, the algorithm



(a) Two (of three) simplified color bands. (b) caterpillar DAGs (G-x) (c) Final caterpillar DAGs and segments (S-$x\,y$).

Figure 5: Caterpillar scheduling for a simplified Montage workflow. (Only selected task vertices are shown; and only initial input data.) Yellow edges denote critical path; dashed blue edges, data reuse. caterpillar DAGs are labeled G-x; segments S-xy.

gathers (i) the set of flow edges E_P for each producer relation $\mathcal{P}_r$ connected to the next level $l+1$, (ii) the set of flow edges E_C for each consumer relation C_r from data at $l+1$ to tasks at level $l+2$. The maximum cardinality of these sets dictates how many tasks should be launched in parallel for maximum achievable concurrency. The sets also dictate the total required storage capacity and maximum achievable flow intensity for the current segment (Line 8).

The growth decision is now made. Using each $E_P \in \mathcal{P}_r$ and $E_C \in C_r$, the algorithm calculates the total required compute resources, storage capacity, and flow intensity. It then determines how many of these *compound* edges (i.e., producer-data-consumer) can be scheduled on one compute node (a) for *vertical* growth or (b) for *horizontal* growth, without violating either a node's storage capacity or maximum available flow rate. Let us assume that the number of such edges is t. These edges and their associated tasks will form one caterpillar segment.

Since growth decisions are made incrementally, subsequent iterations of the algorithm continue to either augment or end an existing segment. Once required resources exceed a node's storage capacity or compute/flow benefits, growth ends (Lines 20–21) and a new segment is created.

Scheduling strategies. After partitioning each caterpillar DAG into segments, the algorithm then performs a trade-off analysis between two scheduling strategies (line 14). The *maximize parallelism* strategy spreads segments across multiple nodes to execute with maximum concurrency. The *maximize flow locality* strategy assigns all segments to the same compute node, scheduling them sequentially. In both cases, the segment's costs account for requisite data flow, including any flow from segment-local storage to a downstream location.

Data reuse. Uniquely, our algorithm accounts for data reuse (Line 13) *between* and *within* tasks in its cost-benefit analysis. Reuse *within* tasks is captured by the DFL's *footprint* metric that distinguishes unique data vs. accessed data. Understanding reuse is important for accurately estimating the impact of the flow-locality strategy. With accurate estimates, the algorithm selects the appropriate growth and scheduling strategy accordingly (Lines 15–18).

Complexity. Assume there are a total of c caterpillar DAGs. For each caterpillar DAG, finding the critical flow takes O(V_t), where V_t is the total number of task vertices in $\mathcal{G}$. Next, grouping into the caterpillar segments takes $O(E)$ time, where E is the total number of edges in $\mathcal{G}$. Assignment of resources to all the groups takes $O(ER)$ time, where r is the resource combinations. The time complexity of the algorithm is thus $O(V_t + ER) \approx O(ER)$. Since R is often bounded by 100, in practice complexity is linear in E.

Example. Figure 5 shows an example for Montage, a workflow that constructs one large image from many small ones while processing color bands (cf. Section 6.2). The figure shows, for two (of three) color bands, the DFL graph (Figure 5a), caterpillar DAGs (Figure 5b), and caterpillar segments (Figure 5c) that result from scheduling.

For caterpillar DAGs, Algorithm 1 first identifies 3 caterpillar DAGs, each corresponding to a color band. Figure 5a shows two caterpillar DAGs (marked as `G-0` and `G-1`). The caterpillar DAG for the other band is identical.

For caterpillar segment decomposition, Algorithm 2 next finds caterpillar segments where, for each caterpillar segment, its tasks/-data are assigned to execute using a single compute node's best matching flow/storage. As shown in Figure 5b, segmentation of `G-0` results in segments `S-00`, `S-01` and `S-02`. `S-00` and `S-01` are instances of horizontal segmentation. The trade-off analysis then decides that `S-00` and `S-01` will be scheduled to maximize parallelism, i.e., distributed across two compute nodes with required flow resources. `S-02` is an instance of vertical segmentation, spanning across multiple levels of the caterpillar DAG, capturing a series of dependent tasks that benefits from co-location.

As an example of data reuse, the data generated by the producer (`mProject`) at stage one is consumed by the `mDiffFit` tasks in the second stage, as well as the `mBackground` tasks at forth stage (dashed blue edges in Figure 5). Capturing the input data reuse information by observing the out-edges from a source vertex to vertices at a later stage (i.e., edges between `mProject` and `mBackground` tasks) allows our algorithm to consider the feasibility of retaining data for reuse during trade-off analysis and yields better assignments.

5 Flow Tradeoff Analysis

This section describes our tradeoff analysis to make the scheduling and assignment decisions for each caterpillar segment. Specifically, the tradeoff analysis combines (a) a rich representation of individual data flows as input passed to (b) performance modeling of data flows based on the Empirical Roofline Tool for I/O (ERT4IO) [52].

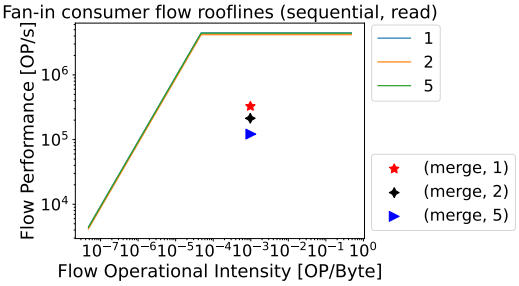


Figure 6: Modeling flow performance for the `merge` (consumer) step in 1000 Genomes workflow (fan-in pattern). Markers show *consumer flow* performance at different per-node concurrencies (1, 2, or 5); each roofline represents maximum performance for the corresponding I/O operation and concurrency.

This allows us to model, compare, and project the performance of distinguished flow (I/O) types/size by the tradeoff analysis. In scheduling, each combination of storage resource and I/O pattern forms one row of the scheduler's resource matrix (Algorithm 2).

ERT4IO. ERT4IO is an empirical I/O Roofline model for data-intensive workload analysis that models the correlation between I/O operations per second (operational performance) and I/O operations per byte (I/O operational intensity). This allows for I/O performance analysis of concurrent/distributed flows that guide tradeoff analysis to make (a) scheduling decisions that yield the best flow parallelism and (b) resource assignments that increase flow rates by selecting the right combination of compute units, data flow, and storage. ERT4IO takes I/O characteristics as input, tested on IOR benchmarks with a number of different specified processes (flow concurrency), providing key insights into parallel I/O performance and contention.

Collecting flow characteristics. We collect each caterpillar segment's task and flow characteristics using the DFL, which are passed to ERT4IO as input to generate insights into flow locality/concurrency and storage options. Particularly, the flow characteristics to collect include flow type (sequential or random), direction (read or write), count (number of operations), and bytes (amount of data accessed). Further, we consider both individual flow edges and entire paths. Models for data flows along paths are constructed from straightforward composites of individual flows along the entire path using sums and averages.

Modeling. We group flows by direction (read or write) with ERT4IO roofline modeling. It is important to separate read (consumer) and write (producer) flows because in-flows and out-flows are semantically different: *reads* are blocking; *writes* are non-blocking *but may block with subsequent reads. Writes* (when blocking) usually take longer than *reads* to complete within storage systems and have different contention profiles. Some *consumers* may re-read data or only read subsets of the total dataset; their access patterns can range from contiguous (ordered data), chunked (HDF5 slices), to random (databases).

Projecting. To make scheduling and assignment decisions we project performance of a proposed flow concurrency using regression analysis based on actual flow performance. Note that our projections are not designed to be exact but sufficiently good to guide the scheduler's decisions. First, with the collected I/O statistics, we can plot the performance curve of the actual flows with different concurrency using the roofline model. The performance curve can signal insights into the trend of performance with respect to different flow concurrency. Using the performance curve, we can project performance (OP/s) for a proposed flow concurrency with different operational intensity (OP/Byte), which can help make assignment decisions on concurrency/locality. For each storage resource, we also include a correctness check to assure that data fits with capacity of that resource.

The coordinate is easily calculated from the individual flows:

- x value (OP/byte): total accesses (for all concurrent flows) vs. access size
- y value (OP/s): total accesses (above) vs. actual time

Second, using the flow roofline, we project performance for a proposed flow concurrency and locality.

- x value (OP/byte): Given the new flow (derived from aforementioned scaling rules), calculate the new ratio of total accesses (OPs) vs. access size (bytes).
- y value (OP/s): As we already have total accesses, the key is a new time. We obtain a new time by scaling the current time with the closest matching roofline (by pattern/concurrency).

Example. Figure 6 shows an example performance model for 1000 Genomes's `merge` consumer flows on **local ramdisks**. This fan-in consumer has 10 `merge` tasks (at input 1000). The tradeoff analysis considers different node parallelisms (2, 5, or 10 nodes), resulting in three per-node flow concurrency options: 1, 2, and 5 (tasks per node). Although the I/O rooflines show little difference, the markers for the *flow pattern* suggest that performance degrades with higher concurrency. We can project the performance by regression with the markers. The regression implication suggests that performance degrades with higher concurrency. We can easily project the performance of concurrency 10, which is likely performing worse than concurrency 5. The implication suggests that the 10 `merge` tasks should be scheduled with 1 task/node. Further, the flow performance could benefit from more powerful compute (for higher OP/s). The improved performance of FastFlow on 10 nodes (Figure 10) shows the effectiveness of the flow models.

Insight. The flow roofline modeling yields valuable insights into optimal flow parallelism, resource assignment decisions, and flow resource (i.e., peak performance and bandwidth).

6 Evaluation

This section demonstrates the impact of FastFlow on distributed scientific workflows. We apply our methodology to four representative workflows, namely Montage (Section 6.2), 1000 Genomes (Section 6.3), DeepDriveMD (Section 6.4), and SRA Search (Section 6.5). The workflows are complex, range from compute to data intensive, and are used as baselines. We perform experiments with input datasets of different sizes and also vary the number of compute resources. We compare our approach with three relevant baseline techniques discussed in Section 6.1 and demonstrate geo-mean speedups of 1.15×, 3.5×, 1.04×, and 1.07× compared to next best (*not* linear time); and up to 1.28×, 87×, 1.4×, and 5× compared to popular linear-time methods. Our evaluation uses the machines in Table 1.

Table 1: Machine configurations for experiments.

Machine	Compute, Memory	Storage options (notes)
CPU CLUSTER 1	12× Intel SkyLake Xeon 6126 @2.60GHz; 192 GB	NFS (default); Lustre; SSD (node); Ramdisk (node)
CPU CLUSTER 2	32× AMD EPYC 7543 @2.8GHz; 256 GB	NFS (default); BeeGFS (w/ caching); SSD (node); Ramdisk (node)
GPU CLUSTER	32× AMD EPYC 7502 @2.5GHz; 6x NVIDIA A100; 256GB	NFS (default); BeeGFS (w/ caching); SSD (node); Ramdisk (node)

Table 2: Comparison of scheduling approaches for experiments.

Approach	Strategy	Complexity (V: vertices; E: edges)
DAGP/METIS	Minimize edge cuts among acyclic partitions	$O(V(V+E))$
CPOP,HEFT	Prioritize tasks on critical paths, tasks with earliest finish time	$O(E)$
FAASFLOW	Prioritize critical paths & patterns	$O(EI)$ (Iterations)
DFMAN	Maximize I/O bandwidth	$O(CSTD)^{3.5}$ (Compute, Storage, Tasks, Data)
FASTFLOW	Optimize critical flows (caterpillars) scheduling	$O(E)$

6.1 Baseline methodology

We compare FastFlow with several relevant baseline methodologies. Table 2 summarizes each method and its complexity. Observe that FastFlow is linear time compared to the second-best methods (FaaSFlow or DFMan).

dagP/METIS. The directed acyclic graph partitioning (dagP) [5, 29] approach partitions a given graph while minimizing the total weights of the edges with endpoints in separate partitions. Compared to the well-known METIS partitioner [33], dagP operates on a directed graph and ensures that different partitions maintain acyclic dependencies among them. For this approach, we assume that datasets are hosted on the Parallel File System (PFS). The result is a baseline optimized by PFS whole-file data movement.

FaaSFlow [39], Critical-Path-on-a-Processor (CPOP), Heterogeneous-Earliest-Finish-Time (HEFT). FaaSFlow [39] prioritizes tasks and data locality along critical paths. Additionally, it includes a few simple fixed graph patterns for context. Note that FastFlow's caterpillar flows include the same context but are comprehensive and based on a general and abstract method. The CPOP [49] and HEFT [49] schemes prioritize based on variant definitions of critical path. Frequently, FaaSFlow's schedules are better than either because data movement usually dominates the time of short tasks. We therefore omit CPOP and HEFT.

DFMan. DFMan [16] assigns data to storage resources based on data consumers (data-to-task edges) in the flow graph, ordered by topological levels. The objective is to maximize overall I/O bandwidth. The first iteration assigns one consumer per data file. Subsequent iterations assign additional consumers to where its data resides. To compare with DFMan, files are placed either on the fastest node-local storage for non-shared data or on the shared file system if multiple processes from different compute nodes require data sharing (including intermediate data) to avoid data copies. Our approach differs not only by minimizing response time, but also by scheduling the combination of critical flows and their interacting flows, according to both flow locality and parallelism.

6.2 Case study: Montage

Montage [4, 30] is a compute-intensive image processing toolkit that combines a collection of astronomical images into composite images (mosaics). As compute intensive, Montage is a challenge for any methodology that emphasizes data flow. Given many small image files (FITS format), Montage stitches them together into a large re-projected and reorientation image.

Configurations. We consider three different resolutions of `fits` image files to vary input sizes: 720x720 (`small`), 3600x3600 (`medium`), and 7200x7200 (`large`) pixels. Over the 3 color bands, we obtain 192 `fits` image files from the space telescope archive (STScI) ranging from 2GB to 20GB in total input data sizes. We conduct experiments on CPU CLUSTER 1 (Table 1).

Scheduling. Figure 5 shows the DFL graph, caterpillar DAGs (caterpillar DAG), and caterpillar segments that result from FastFlow's scheduling and assignment. FastFlow identifies 3 caterpillar DAGs, one for each of three color bands: red (`G-0`), blue (`G-1`), and green. The caterpillars indicate an opportunity for independent execution of each interrelated flow and then, for the final combined output, flow parallelism. Within each caterpillar, the tradeoff analysis identifies multiple horizontal segments (`S-00`, `S-01`) for the first two stages (`mProject` and `mDiffFit`) that achieve the best combination of parallelism and data locality. Additionally, the vertical segment (`S-02`) recommends co-location of the producer-data-consumer flow edges for *multiple subsequent stages* (spanning from `mConc` stage to the final stage `mView`). Moreover, keeping the data generated at a stage (`mProject`) on node-local storage for reusing at a later stage (`mBackground`) is endorsed.

Results. Results comparing FastFlow's recommendation with other methods are shown in Figures 8a and 8b with the Digitized Sky Survey (DSS2) data. Figure 7 shows the breakdown of the total execution time for each stage on two compute nodes with the large data size (7200x7200). As can be observed, FastFlow results in a *geo-mean of 1.1x speedup for strong scaling medium dataset* (Figure 8a) and *a geo-mean of 1.2x speedup with different dataset sizes compared to dagP* (Figure 8b). The performance gain can be attributed to three things. First, co-location of the producer-consumer tasks — indicated by caterpillar segments `S-00` and `S-01` (Figure 5b) — *assigns flows to the best resource*. Second, the flow of several producer-consumer stages (`S-02`) is increased (within storage constraints), compared to the global file system. Third, the segments enable *retention and reuse of data* generated at the `mProject` stage by the later `mBackground` stage.

Figure 8a varies compute nodes for the medium dataset. Figure 8b varies data sizes on 16 nodes. FaaSFlow is typically second. Unlike FaaSFlow, FastFlow *maximizes rates along critical flows*. FastFlow's largest improvements occur for (a) larger datasets and (b) with more compute nodes. Both trends can be explained with data flows. For example, in Figure 8a the notable performance improvements are obtained with more nodes (12.5% and 12.4% on 16 and 8 nodes respectively) as it localizes flows efficiently without causing I/O contention.

6.3 Case study: 1000 Genomes

1000 Genomes [1, 17] is a data-intensive bioinformatics workflow. The workflow processes multiple chromosomes simultaneously

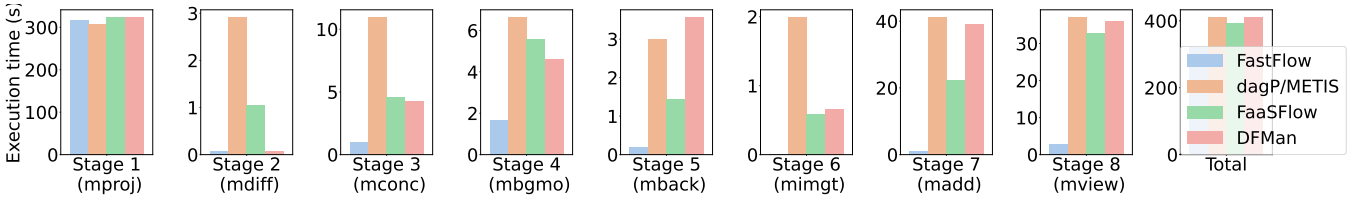
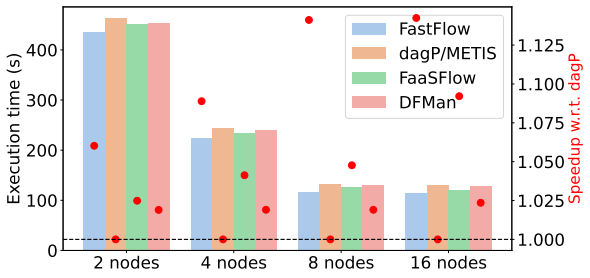
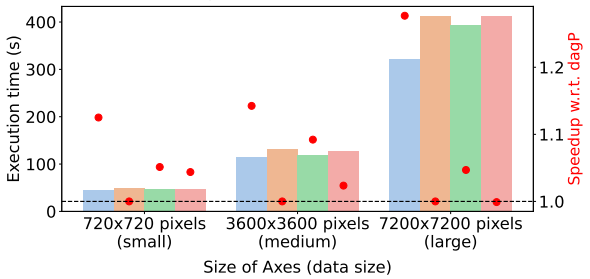


Figure 7: Montage: Impact of scheduling methods on per-stage execution times for large dataset on 16 nodes.



(a) Varying nodes for medium dataset.



(b) Varying datasets on 16 nodes.

Figure 8: Montage: Impact of scheduling methods while varying nodes and dataset size, shown as execution time (bars) and speedup vs. dagP (red dots). Horizontal line: dagP speedup = 1.0.

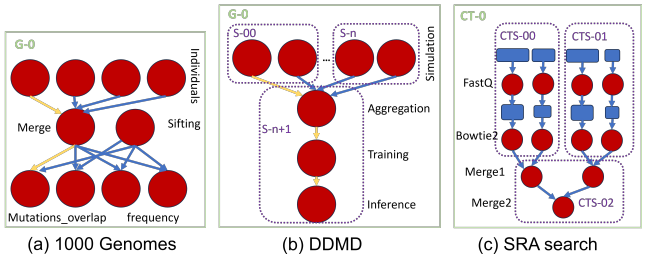


Figure 9: FastFlow decomposition.

using five distinct tasks with varying data split and aggregation patterns.

Configurations. We consider three different input sizes for 1000 Genomes, namely `1000`, `3000`, and `6000`. For all these datasets, 300 `individuals`, 10 `individuals_merge`, 10 `sifting`, 70 `frequency` and 70 `mutation_overlap` task instances were deployed. All experiments were conducted on CPU CLUSTER 2 (Table 1).

Scheduling. Figure 9(a) shows the DFL graph and the caterpillar DAG (grove) that results from FastFlow's scheduling and

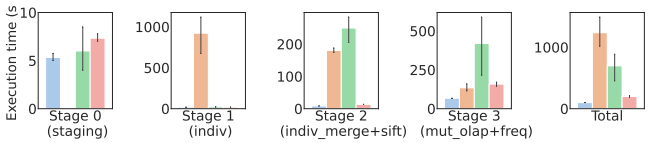
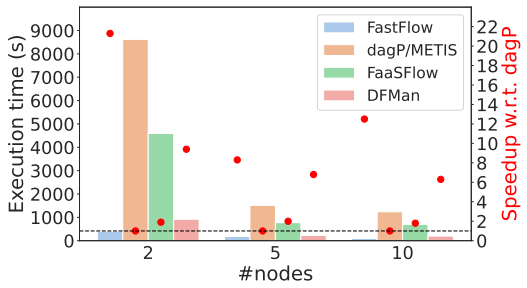
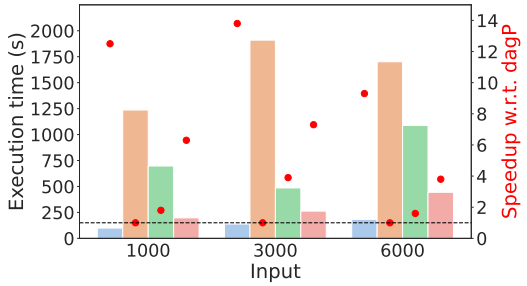


Figure 10: 1000 Genomes: Impact of scheduling methods on per-stage execution times, 10 nodes, input `1000`. Same legend as Figure 11.



(a) Varying nodes for the input `1000`.



(b) Varying inputs on 10 nodes.

Figure 11: 1000 Genomes: Impact of scheduling methods while varying nodes and dataset size, shown as execution time (bars) and speedup vs. dagP (red dots). Horizontal line: dagP speedup = 1.0.

assignment. FastFlow places an entire pipeline for each individual chromosome in a separate caterpillar DAG. The caterpillar DAG indicates an opportunity for independent execution of interacting critical flows, one for each of 10 chromosomes (one shown). The tradeoff analysis determines that each flow should be scheduled on a distinct compute node to *maximize parallelism, data locality, and flow rates*. The new schedule finds the best assignment that collocates multiple I/O metadata flow while avoids I/O contention (cf.Section 5 for details).

Results. The results are shown in Figures 10, 11a and 11b. Figure 10 shows the per-stage breakdown with various scheduling

methods for comparison on 10 nodes with `1000` chromosomes. The execution times vary substantially depending on the method. FastFlow consistently outperforms all the baseline methods in all setups, while DFMan often ranks as the second-best option. This outcome is expected, as data flow is more important when compared to Montage. Note that dagP/METIS runs on PFS entirely. Even though it doesn't have data staging overhead, its performance suffers. The FastFlow-based executions show 7× and 2.4× speedup compared to FaaSFlow and DFMan, which are state-of-the-art scheduling approaches. The key reason is that FastFlow *maximizes locality along critical flows*. That means all data, including inputs and subsequent intermediate files, are on the fastest storage (RAM disks), reducing movement costs and maximizing flow rates.

Figure 11b varies the input sizes and reports execution time and speedup of different methods w.r.t. dagP/METIS on 10 compute nodes. Even when the dataset size increases, FastFlow still shows great performance improvement. This is because a larger performance benefit is expected with larger input sizes as long as the data can fit into node-local storage. In all cases, FastFlow outperforms other scheduling approaches. Specifically, compared to the FaaSFlow and DFMan approaches, FastFlow-based execution runs 7× and 2.4× faster, respectively.

Figure 11a reports the experimental results for strong scaling with `1000` as input. The goal is to demonstrate the efficiency and effectiveness of the flow resource model and tradeoff analysis. The scheduling on 10 nodes, explicitly encouraged by FastFlow, outperforms the executions on 2 and 5 nodes. *Overall, performance improves by up to 87× and 15.3× compared to the baselines on 2 and 5 nodes (discouraged by FastFlow).*

6.4 Case study: DeepDriveMD

DeepDriveMD is a deep learning-driven molecular dynamics simulations workflow for protein folding [3, 36]. It consists of a four-stage pipeline: simulation, aggregation, training and inference. The default configuration targets throughput performance with large chunks of work and long run times. We are interested in improving response time.

Configurations. We consider three simulation time steps for molecular dynamics to vary input data sizes: 100 picoseconds (ps), 500 ps, and 1000 ps as small, medium, and large, respectively. We perform DeepDriveMD experiment on the BBA protein folding system using GPU CLUSTER (Table 1) with CUDA in PyTorch for the ML model.

Scheduling. Figure 9(b) shows the DFL graph and caterpillar segments that result from FastFlow's scheduling and assignment. Simulation tasks are grouped according to resource constraints and will be executed on multiple nodes (`S-0 ... S-n`). Once the simulations complete, the subsequent aggregation, training, and inference tasks execute in a single caterpillar segment (`S-n+1`). FastFlow places data on faster on-node memory as much as possible, while eliminating the need for committing to the PFS after each stage and fetching data from the file system.

Results. Figure 12 shows the performance breakdown of the various DeepDriveMD stages for medium dataset (500ps simulation length) using 4 nodes. FastFlow executions are faster in all setups. In terms of total execution time, FastFlow outperforms all other

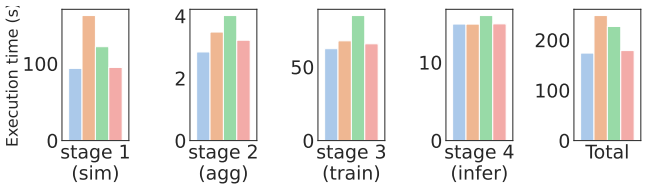
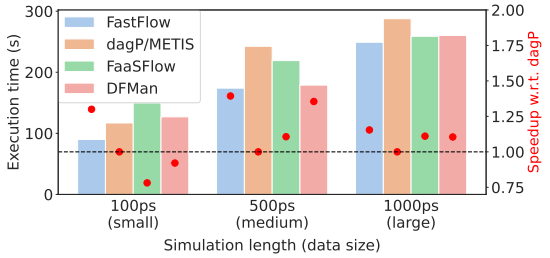
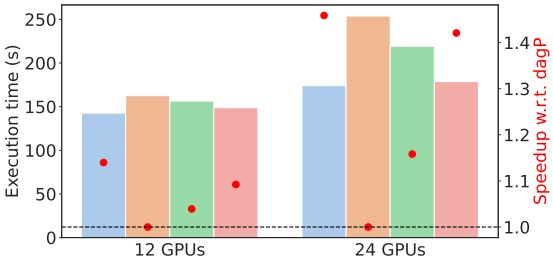


Figure 12: DeepDriveMD: Impact of scheduling methods on per-stage execution times, medium dataset, 4 nodes. Same legend as Figure 13.



(a) Varying datasets size on 4 nodes.



(b) Varying GPUs for medium dataset.

Figure 13: DeepDriveMD: Impact of scheduling methods while varying dataset size and GPUs, shown as execution time (bars) and speedup vs. dagP (red dots). Horizontal line: dagP speedup = 1.0.

methods, as evident from Figure 13a and Figure 13b, where we vary the input dataset size and resources respectively. For both of these Figures, the improvement is reported with respect to the dagP/METIS method. From Figure 13a, it can be observed that, *compared to dagP/METIS, FastFlow achieves 1.3×, 1.4×, and 1.2× speedup for small, medium, and large instances of datasets respectively.* The relative improvements are diminished as data size increases (28% for medium size and 13% for large size). This is due to the fact that longer simulation time (39 seconds out of 89 seconds, 42% of the total execution time with the small data size; whereas 131 seconds out of 241 seconds, 53% of the total execution time with the large data size) in the simulation stage accounts for most of the total execution time. Still, the data flow size from the simulation stage (producer) is not large enough to fully benefit FastFlow's scheduling and assignment. We expect consistent improvement with a small dataset running in an iterative version of the workflow instead of running it with a single large dataset run.

Figure 13b reports the speedups achieved by FastFlow while varying compute resources (2 and 4 compute nodes, each node has 6 GPUs) with DeepDriveMD medium dataset. FastFlow shows the

best speedup against all the other methods ranging from 1.14× to 1.46×, in which we observe better improvements with more nodes. We observe that, *compared to the second-best approach, DFMan, FastFlow achieves a geo-mean of 1.04× speedups across all different experimental settings*.

6.5 Case study: SRA Search

SRA Search is a genomics workflow for aligning sequences to a reference genome [38]. The pipeline downloads (`fasterq-dump`) several FASTQ sequence files from the NCBI archive [9] and performs per-file alignment (`bowtie2`) to find similar characters between the sequence reads and the reference. Results are collected (`merge`) into a single archive.

Configurations. We consider three groups of input data based on read counts of individual sequencing files: less than 1.4 million (`small`), 2.7 million (`medium`), and 5.2 million (`large`). For each group of inputs, 192 SRA files are downloaded from NCBI and distributed across CPU CLUSTER 1, where input sizes total 57GB, 122GB, and 236 GB for `small`, `medium`, and `large`, respectively.

Scheduling. Figure 9(c) shows the DFL graph and the segments identified by FastFlow. In contrast to other workflows, file sizes significantly vary for individual input files in the first and second stages (`FastQ` and `Bowtie2`). The other baselines fail to recognize this important aspect when distributing tasks from the same *stage* with non-uniform input files in size. Both dagP and FaaSflow effectively select a *block* distribution, over-burdening local I/O systems by placing large files on the same flow resources. DFMan improves over these two, but prioritizes global bandwidth rather than the whole critical flow. In contrast, based on models and prioritization of critical flows, FastFlow creates an assignment with an approximate cyclic data distribution: first the largest files are evenly distributed across flow resources, then the next set, and so on.

Results. Figure 14 shows the execution time breakdown for each stage with a small dataset using 8 nodes, resulting in the quickest execution time in total. Figure 15a and Figure 15b report the speedups achieved by FastFlow as we vary the inputs and compute resources, respectively.

Figure 15a shows speedups while varying compute nodes with the `small` dataset. FastFlow achieves the best performance, 4.3×, 4.2x, 2.79x, and 2.8x speedups respectively. In Figure 15b, *FastFlow is 2.2×, 4.9×, and 2.4× faster* compared to dagP/METIS, because of overall increases in flow rates.

Besides its cyclic task-data distribution, there are additional reasons for FastFlow's improvement: (a) the access pattern of each SRA input file is random, (b) the I/O request size varies significantly, (c) as the problem size is scaled, so does the number of I/O accesses and the size of the accesses. These characteristics imply that SRA search's data flow is latency-bound instead of bandwidth-bound. FastFlow's models of flow profiles and performance enable it to reason about the impact of these characteristics on each proposed schedule. FastFlow's schedule balances not only flow payload, but also *flow requests* across resources, improving the imbalance by 15% compared to other baselines. Overall, compared to the second-best approach (DFMan), *FastFlow achieves a geo-mean speedup of 1.07×* across all different experimental settings.

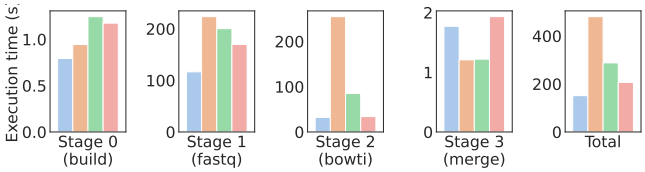
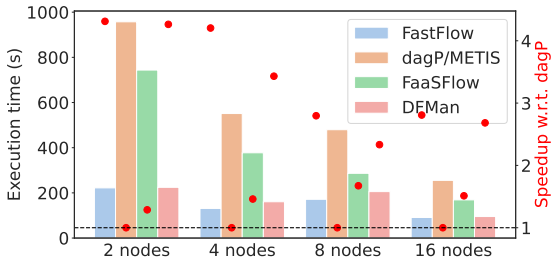


Figure 14: SRA Search: Impact of scheduling methods on per-stage execution times, `small` dataset, 8 nodes. Same legend as Figure 15.



(a) Varying nodes for `small` dataset.

(b) Varying input dataset size on 16 nodes.

Figure 15: SRA Search: Impact of scheduling methods while varying nodes and dataset size, shown as execution time (bars) and speedup vs. dagP (red dots). Horizontal line: dagP speedup = 1.0.

7 Related work

In contrast to data flow lifecycle analysis [35], our primary contributions include: (a) an automated two-tier caterpillar partitioning driven by caterpillar DAGs; (b) model-driven workflow scheduling and resource allocation aimed at enhancing data flows and reducing time-to-solution; and (c) showcasing the significance of concurrently analyzing both the entire caterpillar DAG and its dependent segments.

Data-centric Workflow Orchestration. Extensive research has been conducted on data-centric workflow orchestration to reduce footprints while maximizing locality and data reuse.

Select work includes: *a) Compiler/program optimization* [8, 10, 15, 48, 51], which progressively translates the high-level programming languages to low-level machine code while leveraging program semantics and attributes to apply optimization techniques like vectorization, loop tiling, kernel fusion, and prefetching for better data locality and reuse.

b) Cloud/serverless computing [11, 31, 37, 39–41], which enables data and resource locality-/reuse-aware techniques/optimizations, such as caching, while coordinating tasks over cloud/serverless

workflows to minimize data transmission; and *c) Distributed HPC* (FastFlow is along this line), which schedule dependent task and data flows to the same location to maximize parallelism, data locality, and reuse.

Locality. Locality has been a primary focus for workflow scheduling in HPC systems [16, 21, 28]. One notable work is DFMan [16], which allocates workflow tasks to storage resources to maximize global storage bandwidth. Besides complexity (Table 2) there are three differences compared to our approach. First, DFMan optimizes bandwidth rather than response time, potentially improving bandwidth of *non*-critical paths and *delaying* critical ones. Secondly, DFMan incrementally considers consumer *edges* in a topological order, preventing it from reasoning about *multiple* flow *paths* simultaneously. In contrast, FastFlow considers *multiple paths*, where each path can include many producers and consumers. Additionally, FastFlow assesses whether each path group (segment) benefits more from flow locality or parallelism. Finally, FastFlow uses models of data flow that (a) capture both inter- and intra-task data locality, and (b) distinguish between latency-bound (small random reads), bandwidth-bound (large sequential writes), and contention-bound (concurrent small operations).

Reuse. Data reuse is essential for minimizing data movement from remote locations in distributed large-scale systems [19, 35, 47]. DAGuE [12] develops a workflow DAG scheduling scheme with MPI for distributed computing, which uses a greedy strategy to maximize parallel tasks on a NUMA node to improve locality and data reuse. Pegasus [19] leverages a data catalog to capture input/output and intermediate data staging for data reuse.

DAG-informed scheduling. Graph analysis such as critical paths and min-flow/max-cut has been used for scheduling.

Critical path. Approaches prioritizing response time often utilize different versions of the DAG's critical path [7, 14, 42]. Noteworthy algorithms include Heterogeneous Earliest-Finish-Time (HEFT) and Critical-Path-on-a-Processor (CPOP) [49], and their various adaptations [33, 50]. HEFT schedules tasks by prioritizing those with the earliest possible finish time, factoring in the task cost and the subsequent critical path. Conversely, CPOP determines priorities based on the entire critical path that intersects with a given task. Lee et al. [35] pinpoint scheduling bottlenecks through data flow analysis and DAG evaluation but do not propose scheduling.

Most relevant is FaaSFlow [39], a serverless scheduler for task DAGs (workflows) that prioritizes critical-path locality and efficient resource use. FaaSFlow schedules critical paths, assigned to DAG partitions to the same worker to maximize locality, until resource contention occurs. Its greedy and iterative partitioning strategy has non-linear complexity. In contrast, FastFlow can craft faster schedules using a linear-time algorithm because it may delay interactions along the critical flow, which FastFlow captures using its caterpillar partitions. By adopting a global view through caterpillar DAGs, FastFlow achieves rapid and efficient segmentation. Additionally, FastFlow employs a model-driven approach to prioritize critical flows, balancing between locality and parallelism.

Min-flow/max-cut. Most scheduling methods that address the min-flow/max-cut problem are NP-complete [25]. One widely-used technique, METIS [32, 33], aims to minimize communication costs in distributed parallel numerical simulations. Similarly, dagP [29] identifies regions of many dependent flows, which tends to prioritize throughput over path response time.

8 Conclusions

This paper presented FastFlow, a novel methodology to improve workflow response times by improving a workflow's data flow via scheduling and resource assignment. Our evaluation shows best-in-class performance (1.15×, 3.5×, 1.04×, and 1.07× compared to the next best) across both compute and data-intensive workflows. Further, FastFlow's scheduling is linear time in comparison to the next-best alternatives (Table 2).

We conclude that when optimizing a workflow's response time, it is important to understand critical flow paths and their interacting in/out flow paths. That is, the global perspective of caterpillar partitions and segments, based on interactions that may *delay* or *shift* the critical path, has a unique ability to guide fast scheduling algorithms. We further highlight three additional observations. First, scheduling and resource assignment using caterpillar DAGs and segments can effectively reduce response time. Second, these groves and segments enable reasoning about beneficial flow *parallelism* and *locality*. Finally, tradeoff analyis based on data flow statistics for flow paths enable reasoning about data flow bottlenecks at both the granularity of entire paths and individual producer-consumers.

Acknowledgments

This research is supported by the U.S. Department of Energy (DOE) through the Office of Advanced Scientific Computing Research's "Orchestration for Distributed & Data-Intensive Scientific Exploration" and the "Decentralized Data Mesh for Autonomous Materials Synthesis" AT SCALE LDRD at Pacific Northwest National Laboratory. PNNL is operated by Battelle for the DOE under Contract DE-AC05-76RL01830.

References

[1] [n. d.]. 1000Genomes Workflow Git repo. https://github.com/pegasus-isi/1000genome-workflow. Accessed: 2023-03-15.
[2] [n. d.]. Caterpillar tree. https://en.wikipedia.org/wiki/Caterpillar_tree. Accessed: 2023-03-15.
[3] [n. d.]. DeepDriveMD Workflow Git repo. https://github.com/radical-collaboration/DeepDriveMD. Accessed: 2023-03-15.
[4] [n. d.]. Montage Workflow Git repo. https://github.com/wfcommons/pegasus-instances/tree/master/montage. Accessed: 2023-03-15.
[5] [n. d.]. Multilevel Directed Acyclic Graph Partitioner (dagP). https://github.com/GT-TDAlab/dagP/tree/master. Accessed: 2023-08-15.
[6] 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 Intl. Conf. on e-Science*. 32–43.
[7] Khaled Almi'ani and Young Choon Lee. 2016. Partitioning-Based Workflow Scheduling in Clouds. In *2016 IEEE 30th Intl. Conf. on Advanced Information Networking and Applications (AINA)*. 645–652. doi:10.1109/AINA.2016.83
[8] Riyadh Baghdadi, Jessica Ray, Malek Ben Romdhane, Emanuele Del Sozzo, Abdurrahman Akkas, Yunming Zhang, Patricia Suriana, Shoaib Kamil, and Saman Amarasinghe. 2019. Tiramisu: A polyhedral compiler for expressing fast and portable code. In *2019 IEEE/ACM Intl. Symp. on Code Generation and Optimization (CGO)*. IEEE, 193–205.
[9] Tanya Barrett, Stephen E Wilhite, Pierre Ledoux, Carlos Evangelista, Irene F Kim, Maxim Tomashevsky, Kimberly A Marshall, Katherine H Phillippy, Patti M Sherman, Michelle Holko, et al. 2012. NCBI GEO: archive for functional genomics data sets—update. *Nucleic acids research* 41, D1 (2012), D991–D995.
[10] Tal Ben-Nun, Johannes de Fine Licht, Alexandros N Ziogas, Timo Schneider, and Torsten Hoefler. 2019. Stateful dataflow multigraphs: A data-centric model for performance portability on heterogeneous architectures. In *Proc. of the Intl. Conf. for High Performance Computing, Networking, Storage and Analysis*. 1–14.

[11] Marcel Blöcher, Lin Wang, Patrick Eugster, and Max Schmidt. 2021. Switches for HIRE: Resource scheduling for data center in-network computing. In *Proc. of the 26th ACM Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*. 268–285.

[12] George Bosilca, Aurelien Bouteiller, Anthony Danalis, Thomas Herault, Pierre Lemarinier, and Jack Dongarra. 2012. DAGuE: A generic distributed DAG engine for high performance computing. *Parallel Comput.* 38, 1-2 (2012), 37–51.

[13] 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 Intl. Parallel and Distributed Processing Symp. (IPDPS)*. IEEE.

[14] Giorgio Buttazzo, Enrico Bini, and Yifan Wu. 2011. Partitioning Real-Time Applications Over Multicore Reservations. *IEEE Transactions on Industrial Informatics* 7, 2 (2011), 302–315. doi:10.1109/TII.2011.2123902

[15] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. 2018. TVM: An automated End-to-End optimizing compiler for deep learning. In *13th USENIX Symp. on Operating Systems Design and Impl.* 578–594.

[16] 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 Intl. Parallel and Distributed Processing Symp. (IPDPS)*. 368–378. doi:10.1109/IPDPS53621.2022.00043

[17] 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).

[18] Ferreira da Silva et al. 2024. Workflows Community Summit 2024: Future Trends and Challenges in Scientific Workflows. doi:10.5281/zenodo.13844759

[19] 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.

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

[21] Naznin Fauzia, Venmugil Elango, Mahesh Ravishankar, J. Ramanujam, Fabrice Rastello, Atanas Rountev, Louis-Noël Pouchet, and P. Sadayappan. 2013. Beyond Reuse Distance Analysis: Dynamic Analysis for Characterization of Data Locality Potential. *ACM Trans. Archit. Code Optim.* 10, 4, Article 53 (dec 2013), 29 pages. doi:10.1145/2541228.2555309

[22] Rafael Ferreira da Silva, Rosa M. Badia, Deborah Bard, Ian T. Foster, Shantenu Jha, and Frédéric Suter. 2024. Frontiers in Scientific Workflows: Pervasive Integration With High-Performance Computing. *Computer* 57, 8 (2024), 36–44. doi:10.1109/MC.2024.3401542

[23] 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.

[24] 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.

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

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

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

[28] Stephen Herbein, Dong H. Ahn, Don Lipari, Thomas R.W. Scogland, Marc Stearman, Mark Grondona, Jim Garlick, Becky Springmeyer, and Michela Taufer. 2016. Scalable I/O-Aware Job Scheduling for Burst Buffer Enabled HPC Clusters. In *Proc. of the 25th ACM Intl. Symp. on High-Performance Parallel and Distributed Computing* (Kyoto, Japan) *(HPDC '16)*. Association for Computing Machinery, New York, NY, USA, 69–80. doi:10.1145/2907294.2907316

[29] 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.

[30] Joseph C Jacob, Daniel S Katz, G Bruce Berriman, John C Good, Anastasia Laity, Ewa Deelman, Carl Kesselman, Gurmeet Singh, Mei-Hui Su, Thomas Prince, et al. 2009. Montage: a grid portal and software toolkit for science-grade astronomical image mosaicking. *Intl. J. of Comp. Science & Engineering* 4, 2 (2009), 73–87.

[31] Hidehiro Kanemitsu, Kenji Kanai, Jiro Katto, and Hidenori Nakazato. 2021. A containerized task clustering for scheduling workflows to utilize processors and containers on clouds. *J. of Supercomputing* 77, 11 (2021), 12879–12923.

[32] George Karypis and Vipin Kumar. 1995. Analysis of Multilevel Graph Partitioning. In *Proc. of the 1995 ACM/IEEE Conf. on Supercomputing* (San Diego, California, USA) *(Supercomputing '95)*. Association for Computing Machinery, New York, NY, USA, 29–es. doi:10.1145/224170.224229

[33] George Karypis and Vipin Kumar. 1998. A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs. *SIAM Journal on Scientific Computing* 20, 1 (1998), 359–392. doi:10.1137/S1064827595287997 arXiv:https://doi.org/10.1137/S1064827595287997

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

[35] Hyungro Lee, Luanzheng Guo, Meng Tang, Jesun Firoz, Nathan Tallent, Anthony Kougkas, and Xian-He Sun. 2023. Data Flow Lifecycles for Optimizing Workflow Coordination. In *Proc. of the Intl. Conf. for High Performance Computing, Networking, Storage and Analysis (SuperComputing)* (Denver, CO, USA) *(SC '23)*. Association for Computing Machinery, New York, NY, USA. doi:10.1145/3581784.3607104

[36] 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.

[37] Pawissanutt Lertpongrujikorn and Mohsen Amini Salehi. 2023. Object as a service (oaas): Enabling object abstraction in serverless clouds. In *2023 IEEE 16th Intl. Conf. on Cloud Computing (CLOUD)*. IEEE, 238–248.

[38] Kyle Levi, Mats Rynge, Eroma Abeysinghe, and Robert A Edwards. 2018. Searching the sequence read archive using Jetstream and Wrangler. In *Proc. of the practice and experience on advanced research computing*. 1–7.

[39] Zijun Li, Yushi Liu, Linsong Guo, Quan Chen, Jiagan Cheng, Wenli Zheng, and Minyi Guo. 2022. Faasflow: Enable efficient workflow execution for function-as-a-service. In *Proc. of the 27th ACM Intl. Conf. on architectural support for programming languages and operating systems*. 782–796.

[40] Zijun Li, Chuhao Xu, Quan Chen, Jieru Zhao, Chen Chen, and Minyi Guo. 2023. DataFlower: Exploiting the Data-flow Paradigm for Serverless Workflow Orchestration. In *Proc. of the 28th ACM Intl. Conf. on Architectural Support for Programming Languages and Operating Systems, Volume 4*. 57–72.

[41] Hadeer Mahmoud, Mostafa Thabet, Mohamed H Khafagy, and Fatma A Omara. 2021. An efficient load balancing technique for task scheduling in heterogeneous cloud environment. *Cluster Computing* 24, 4 (2021), 3405–3419.

[42] Ruben Mayer, Christian Mayer, and Larissa Laich. 2017. The Tensorflow Partitioning and Scheduling Problem: It's the Critical Path!. In *Proc. of the 1st Workshop on Distributed Infrastructures for Deep Learning* (Las Vegas, Nevada) *(DIDL '17)*. Association for Computing Machinery, New York, NY, USA, 1–6. doi:10.1145/3154842.3154843

[43] 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 Intl. Conf. on e-Science*. 284–294.

[44] 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.

[45] National Academies of Sciences, Engineering and Medicine. 2022. *Automated Research Workflows For Accelerated Discovery: Closing the Knowledge Discovery Loop*. The National Academies Press, Washington, DC. doi:10.17226/26532

[46] 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 Intl. Publishing, Cham, 211–223.

[47] Thanh Son Phung, Ben Clifford, Kyle Chard, and Douglas Thain. 2023. Maximizing Data Utility for HPC Python Workflow Execution. In *Proc. of the SC'23 Workshops of The Intl. Conf. on High Performance Computing, Network, Storage, and Analysis*. 637–640.

[48] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *Acm Sigplan Notices* 48, 6 (2013), 519–530.

[49] H. Topcuoglu, S. Hariri, and Min-You Wu. 2002. Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE Transactions on Parallel and Distributed Systems* 13, 3 (2002), 260–274. doi:10.1109/71.993206

[50] Baixuan Wu, Zheng Xiao, Peiying Lin, Zhuo Tang, and Kenli Li. 2023. Critical Path Awareness Techniques for Large-Scale Graph Partitioning. *IEEE Transactions on Sustainable Computing* 8, 3 (2023), 412–422. doi:10.1109/TSUSC.2023.3263172

[51] Yunming Zhang, Mengjiao Yang, Riyadh Baghdadi, Shoaib Kamil, Julian Shun, and Saman Amarasinghe. 2018. Graphit: A high-performance graph dsl. *Proc. of the ACM on Programming Languages* 2, OOPSLA (2018), 1–30.

[52] Zhaobin Zhu, Niklas Bartelheimer, and Sarah Neuwirth. 2023. An Empirical Roofline Model for Extreme-Scale I/O Workload Analysis. In *2023 IEEE Intl. Parallel and Distributed Processing Symp. Workshops*. 622–627. doi:10.1109/IPDPSW59300.2023.00106