Performance Characterization and Provenance of Distributed Task-based Workflows on HPC Platforms

Amal Gueroudji[1], Chase Phelps[2], Tanzima Z. Islam[2], Philip Carns[1],
Shane Snyder[1], Matthieu Dorier[1], Robert B. Ross[1], Line C. Pouchard[3]
[1]*Argonne National Laboratory,* [2]*Texas State University,* [3]*Sandia National Laboratories*
Email:{agueroudji,carns,snyders,mdorier,rross}@anl.gov, {chaseleif,tanzima}@txstate.edu, {lcpouch}@sandia.gov

***Abstract*—Understanding performance and provenance of task-based workflows poses significant challenges, particularly in distributed configurations where resources are shared by multiple applications. Task-based workflow management systems further complicate performance predictability because of their dynamicity that subtly alters task execution order from run to run. In this paper we propose a layered characterization framework for performance and task provenance for Dask.distributed workflows running on high-performance computing (HPC) platforms. It collects data from jobs, the workflow management system, and the operating system to aid in understanding the performance of these workflows. Our approach encompasses three main contributions: first, an extension of Dask.distributed to capture high-fidelity task provenance using Mochi data services; second, the adaptation of the established HPC I/O characterization tool Darshan to gather high-fidelity I/O data, thereby enhancing the granularity of our analysis; and third, a framework to combine and process the collected data and provide helpful insights into performance characterization and reproducibility, alongside our lessons learned.**

***Index Terms*—High-performance computing (HPC), task-based workflows, performance characterization, performance variability, provenance, performance reproducibility, Dask**

I. Introduction

Scientific workflows executed on high-performance, heterogeneous computing environments rely on the complex composition of system software, middleware services, and application codes [1]. Pythonic task-based programming models are increasingly prevalent for large-scale computations and data-intensive processing because of their high productivity. Examples of such tools include Parsl [2], Dask [3], and RADICAL-Pilot [4].

In contrast to traditional MPI-based workflows, which are predominantly static in nature and where the tasks assigned to each MPI process are known in advance, task-based workflows are dynamically scheduled. These workflows rely on a runtime scheduler to efficiently orchestrate resources within a single application. Scheduling decisions are made transparently and dynamically as the workflow progresses, and tasks are dispatched to available resources at runtime, resulting in a lack of a priori information regarding the specific tasks that each worker will execute. While this approach to resource management hides the complexity of the underlying infrastructure and the resource/task management from users, it also obscures a potential source of eventual performance variability.

Reproducibility, defined as the ability to reliably recreate the same results, outputs, or behaviors from a given set of inputs, configurations, and conditions [5], encounters significant challenges in computational sciences because of complicating factors such as non-deterministic ordering of floating-point operations, parallel executions, workflow patterns, and job scheduling.

In this paper we study Dask workflows to identify task behavior, performance, and provenance to determine which tasks, task behaviors, and system characteristics are responsible for the largest variations during multiple executions of the same set of codes in the same configurations. We use these performance variability measures to help us determine the sources of overall irreproducibility in distributed workflow performance. In particular, we aim to answer the following research questions in this work:

1) What workflow management system (WMS) capabilities are required to extract this information?
2) What information from a dynamically scheduled workflow is required to characterize performance variations?
3) What correlations between tasks and events would help us investigate performance variability and understand the sources of latency and irreproducible performance?
4) Can we identify gaps in the metadata collection?

This work addresses these questions for Dask workflows by collecting data from different layers and tools and storing the data in a common tabular format with shared identifiers between the different sources, facilitating compliance with FAIR principles [6], especially interoperability and reusability. Our contributions include the following:

- Data collection at the WMS level: we extend Dask.distributed to extract high-fidelity task provenance by leveraging a Mochi HPC [7] event streaming service known as Mofka. This enables us to track the detailed lineage and execution history of individual tasks without perturbing the workflow system.
- Data collection at the I/O level and correlation with task data: we adapt the well-established HPC I/O characteriza-

tion tool Darshan [8] to gather more granular performance data including detailed insights into task execution, including I/O behavior, with thread-level fidelity.
- Analysis framework: we introduce a framework for dissecting and understanding the performance dynamics of Dask.distributed workflows.
- Lessons learned: we discuss the challenges and lessons learned in this study and identify additional metadata that needs to be collected to explain the gaps.

The remainder of this paper is organized as follows. In Section II we summarize related work, and in Section III we present our proposed implementation with Dask, Darshan, and Mofka. Section III-E discusses provenance and metadata extraction, and Section IV proposes an evaluation including analysis and visualization. In Section V we discuss the challenges and lessons learned during our investigationIn Section VI we summarize our conclusions and briefly discuss ideas for future work.

II. Related Work

Many HPC performance profiling tools are available to aid in understanding and optimizing the performance of scientific applications [9], [10]. Although these tools are highly effective across a wide range of applications, they do not include native workflow support. Specifically, they do not present tasks as distinct entities from threads or processes, nor do they understand the specifics of task execution in a WMS, since these vary by implementation. Thus, it is difficult for these tools to map from low-level performance counters to specific tasks and dependencies, which is crucial for understanding these workflows.

Several task-based workflows offer their own integrated performance profiling tools tailored to the WMS they operate within. However, these tools often lack integration with other profilers or performance characterization tools, limiting their versatility and interoperability. For example, Dask [3] provides a comprehensive dashboard for diagnostics, including task progress, bytes stored by workers, CPU utilization, and other relevant metrics, but this capability is available only to Dask workflows. Similarly to Dask, Ray [11] offers a web-based dashboard with a rich set of data and plots about cluster utilization and resource status, node count and status, and running tasks. Ray also introduces the capability to filter logs using Loki,[1] enhancing workflow behavior understanding. Pegasus [12] provides information about the running workflow and job statistics, integrity metrics (checksum), task status and data, and a notification system for both job and workflows.

Despite the richness of data provided by these tools, extracting and processing this information for further analysis, such as performance reproducibility studies or correlation with data from other tools, remain challenging because the collected data is often specific to each workflow system. This limitation underscores the need for improved interoperability and data exchange mechanisms across different performance profiling tools and between different layers. Reconciling data sources from heterogeneous services (e.g., mapping from low-level performance counters to specific workflow tasks) is a nontrivial step required for analysis and is not resolved with these tools. Before improving the reproducibility of performance in HPC workflows, one needs to be able to measure it at a low level of task or function granularity instead of aggregate statistics, as is commonly done with current performance tools. This work addresses these challenges for Dask workflows by collecting data from different layers and tools and storing the data in a common tabular format, facilitating compliance with FAIR principles, especially interoperability and reusability.

III. Proposed Implementation

In this paper we focus on studying the performance and the provenance of Dask.distributed workflows while running on HPC platforms. Nevertheless, our approach can be used for other workflow management systems and tools.

A. *Dask.distributed workflows*

Dask.distributed [3] is a distributed task-based framework and WMS known for its versatility and productivity. It offers multiple approaches to writing programs, ranging from utilizing lower-level decorators and futures for task creation to leveraging higher-level distributed interfaces for well-known libraries such as NumPy, pandas, and Scikit-learn. In Dask, a task is essentially a function that executes within a distributed process known as a worker, and a workflow is described as a directed acyclic graph, where nodes are tasks and edges are task dependencies. A typical Dask cluster comprises three primary entities: (a) the client is responsible for creating and submitting tasks to a runtime scheduler; (b) the scheduler orchestrates tasks within the cluster, dispatching tasks to available workers and managing their execution; and (c) workers are the computational units responsible for executing tasks, communicating with each other, and storing data.

B. *Aggregating workflow instrumentation with Mofka*

The volume and velocity of telemetry generated at scale have the potential to exacerbate performance challenges for instrumented workflows. We must therefore take care to collect, aggregate, and store this telemetry using lightweight mechanisms. This could be accomplished with a global system-level metrics service, such as LDMS [13], or with a local user-level service that runs in tandem with the workflow. We have elected to employ the latter approach in this study in order to maximize portability. Specifically, we leverage the Mofka[2] event streaming service from the Mochi [7] composable data services framework to manage characterization data.

An event streaming service that runs in tandem with the workflow offers several advantages for this use case. The event streaming model, as pioneered in distributed services such as Kafka [14], is intrinsically designed to ingest large volumes of small but highly concurrent *events*. Events are buffered, aggregated, partitioned, transported, and stored by using a

[1]https://github.com/grafana/loki

[2]https://mofka.readthedocs.io/en/latest/

combination of memory, network, and disk resources to avoid overwhelming producers or consumers. The event streaming model enables our framework to support both in situ analysis and postprocessing analysis. Event streams are persistent data structures, and the API for consuming events is identical whether consumers process events individually in real time or in bulk at the completion of a workflow. This property also means that workflow execution and in situ analysis can each proceed at their own pace, in contrast with more conventional time-multiplexed in situ methods. Moreover, services that are implemented by using the Mochi composable data services framework can be executed in user space without administrative privileges. They can be executed alongside the workflow, on any platform, and scaled as needed for a given workflow instance. We also thus avoid introducing system software dependencies that would require a specialized platform configuration or administrative privileges.

Mofka is a distributed event streaming service, analogous to the more widely known Kafka [14] but optimized specifically for HPC use cases. It leverages HPC architectural features such as high-performance networks, remote direct memory access, high-performance local NVMe storage devices, and high-concurrency multicore CPUs; and it uses an event structure tailored to suit high-volume scientific data. Mofka is constructed using Mochi [7], a methodology and collection of reusable components for rapid development of HPC data services. Mofka uses the following reusable Mochi microservices[3]: Yokan to store key/value data, Warabi to store raw (blob) data, Bedrock for deployment and bootstrapping, and SSG for group membership and fault detection.

Mofka client applications can act as producers or consumers. A producer pushes `events` that are organized into `topics` in the servers. Each event has two parts. The first is a data portion that contains the raw data payload. The second is metadata expressed in JSON format to describe the data. Consumers subscribe to specific topics and pull events from servers to process them. Mofka can be tuned for specific use cases and optimizes transfers using a non-blocking API, background network and processing threads, batching strategies, prefetching, and customizable serialization/deserialization of event metadata. In our framework we treat Dask as the producer (injecting units of characterization data as events into Mofka) and our analysis tools as the consumer (processing and interpreting characterization data to provide insight into reproducibility). More details about how we integrated Dask with Mofka can be found in Section III-E2.

C. Characterizing I/O behavior with Darshan

Collecting task-level performance and provenance data is not always enough for reproducibility analysis. I/O behavior, for example, is a key factor in overall workflow performance. I/O is also known to be a prominent source of performance variability at scale on HPC systems [15], [16]. We have opted to use Darshan [8] to capture workflow I/O behavior because of its minimal overhead, proven usage in the HPC community, and availability of flexible analysis tools [17].

Darshan is a lightweight application-level I/O characterization tool designed to capture the I/O behavior of HPC applications at scale. It collects a plethora of information, including I/O operation counts, access sizes, and cumulative times, and provides a correlation of the I/O operations at the granularity of OS processes and MPI ranks. In this paper we use Darshan eXtended Tracing module (DXT) [18] to collect full tracing of POSIX read/write APIs. We enhance DXT to collect finer-grained information in order to correlate the I/O operations with specific Dask tasks. More detail about this enhancement can be found in Section III-E3.

D. Multisource data analysis and visualization engine

To understand the performance characteristics of different entities of the ecosystem from worker, client, task, and thread levels, we present two types of analysis: (1) single-source and (2) multisource. In a single-source analysis, we retrieve information from a single log and correlate variability in a metric of interest with the factors present within the source. For instance, the Darshan log alone is sufficient to understand I/O characteristics of different threads as a workflow progresses.

However, complexity arises as different components of the composable architecture generate data in diverse formats. For instance, the format of the logs generated by Darshan does not match those generated by Dask, making the data aggregation process challenging. While many performance trace collection tools provide a GUI-based interface to analyze data [9], [19], [20], these tools typically work only with data in a specific format, such as a performance trace file.

To address these challenges in wrangling data collected across various ecosystem layers for extracting insights about how different factors correlate with performance variability, in this paper we present PERFRECUP, a Python-based data aggregation, analysis, and visualization engine. The PERFRECUP engine reads performance data and logs generated by many layers from different tools such as Darshan, Dask, and job schedulers and provides uniform data structures built atop the pandas library [21].

PERFRECUP uses the data (e.g., a view from a specific worker) to focus analyses on regions of interest within all output. Specifically, PERFRECUP combines information from Darshan logs and from Dask scheduler and worker logs, including task keys, dependencies, state transitions, location in the distributed memory (worker, thread), worker communication, and other events from the clients, workers, and scheduler such as the unresponsiveness of the Tornado event loops or garbage collection interruptions, to create pandas DataFrames as "views." For instance, both Darshan and Dask logs contain `pthread ID` and timestamps that can be used to align specific events. The amount of communication and I/O, aligned by execution time, can be compared across several runs to identify whether variability exists and relate variability to task, network, and I/O statistics.

[3]https://mochi.readthedocs.io/

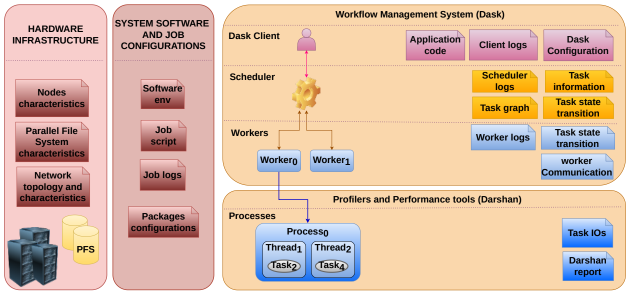

Fig. 1: Example data provenance chart for an HPC workflow. Provenance is collected at the hardware, system software, and application layers. Dask is shown as a representative WMS while Darshan is shown as a representative performance tool.

E. Provenance and metadata extraction

In a typical Dask.distributed workflow, after acquiring the requested resources, the client and workers connect to the scheduler. The client creates and submits task graphs, and the workers run assigned tasks, save, and serve data. In this section we present an overview of the data provenance chart of Dask workflows and then detail the integration of Dask with Mofka and Darshan.

1) Data provenance chart

In this section we present the data provenance chart and detail the need for and impact of the collected data at each layer. In Figure 1 we show the metadata collected at the hardware infrastructure, system software and job configurations, and application layer including the WMS and the profiler components.

In *hardware infrastructure* we collect platform characteristics, including CPU, GPU, SSD, memory, PFS, and network topology characteristics. Then, in *system software and job configurations*, we collect data about the OS, loaded modules, compilers, and installed packages and configurations. We also collect job-level data, including job scripts and logs, to provide insight into the requested and allocated resources. These first two layers introduce the potential for performance unpredictability. For example, the allocated nodes may vary in performance due to factors such as network topology. Moreover, if the Dask scheduler and worker nodes are connected to different switches, some workers may experience increased latency. Furthermore, communication between workers may vary depending on their placement, that is, the worker to node assignment. Thus, we not only incorporate infrastructure, software, and job allocation but also package configuration details, such as Dask's timeouts, heartbeat intervals, and communication settings from the `distributed.yaml` file. In the application layer we collect the data from the workflow management system and profilers and performance tools. In the WMS we collect the client code, the number of task graphs, and eventual configuration changes. Additionally, we collect client logs that may contain warnings or other information about the running workflow. When tasks arrive at the scheduler, we extract all task-related data, such as task keys, groups, prefixes, and dependencies. We also capture all task

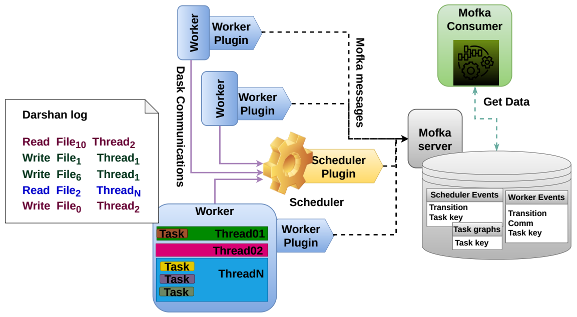

Fig. 2: Dask–Mofka integration.

state transitions, scheduling evolution, and task performance. Here, we keep the scheduler logs, which contain data about the connection/disconnection of the clients and workers, information, warnings, and eventual errors while running tasks. When a task is dispatched to a worker, we gather task state transitions in the worker to identify the time spent in a worker before execution; we also collect communication data between different workers. In addition, we collect worker logs, which may contain information including garbage collection status, and event loop warnings. All data collected at this layer is streamed in situ to Mochi servers for processing.

I/O performance can be a highly variable component of overall workflow performance [22]. We therefore collect I/O counters using an augmented version of the Darshan I/O characterization tool.[4] Task-granular I/O characterization provides a means to evaluate the cost of I/O and the impact of alternative I/O strategies within overall workflow performance.

2) Dask–Mofka plugins

To efficiently gather extensive task provenance data from Dask workflows, we have introduced a new extension to Dask that streams this data via Mofka. We have developed two components serving as plugins for the Dask scheduler and worker classes. These plugins are incorporated into the scheduler and worker launch processes. Their primary function is to intercept specific calls within the classes and extract pertinent data from the ongoing events. For example, during a task state transition, our plugins capture crucial details such as the task key, group, prefix, initial state, final state, timestamp, and the stimuli that triggered this transition. Upon task completion, we retrieve additional information, including the IP address of the worker where the task was executed, the thread ID, start and end times, and the size of the task result. Our plugins facilitate comprehensive tracking and analysis of task behavior and workflow progression within the Dask environment. This information enables users to gain deeper insights into the performance and execution dynamics of distributed workflows. Utilization of this information during runtime to enhance the management of computational tasks is left for future work.

3) Adapting Darshan to enable task-level I/O analysis

We instrument each worker with our modified version of Darshan in order to incorporate I/O instrumentation into our

[4]The Darshan modifications demonstrated in this study will be incorporated into a future Darshan point release.

provenance data. In Dask.distributed, workers execute many tasks within the context of a single POSIX process through the use of an independent thread for each task. Therefore, we extend the DXT module to capture the POSIX thread (pthread) IDs. These can later be correlated with the thread identifier returned by `threading.get_ident()` at the Dask.distributed level. This additional information, along with timestamps, enables us to correlate Darshan records with specific tasks, thereby providing a more detailed understanding of a task's performance and associated I/O operations. We have opted to collect data from Dask and Darshan separately and then fuse them at analysis time to avoid cross-component communication overhead as well as to make the contribution generic and tool-agnostic.

IV. Evaluation

A. Platform and software

We use the Argonne Leadership Computing Facility supercomputer Polaris, which has 560 nodes. Each node has one 2.8 GHz AMD EPYC Milan 7543P 32 core CPU with 512 GB of DDR4 RAM, 4 NVIDIA A100 GPUs, and a pair of Slingshot 11 network adapters. We have used Lustre file systems, each residing on an HPE ClusterStor E1000 platform equipped with 100 petabytes of usable capacity across 8,480 disk drives, with an aggregate data transfer rate of 650 GB/s.

B. Dask workflows

We have collected performance and provenance data from three workflows inspired by Dask examples[5] and its integration with other tools such as Pytorch.[6] The workflows differ from each other in aspects such as data type and size, the type, size, and number of tasks, whether created automatically or manually, and how the task graphs are submitted (step by step or all at once). We have performed 10 runs for the `ImageProcessing` and the `ResNet152` workflows and 50 runs for the `XGBOOST` because it showed more variability. We have used a similar job configuration for all the experiments: 2 worker nodes, 4 workers per node, 8 threads per worker.
`ImageProcessing` pipeline: This workflow consists of a four-step pipeline: normalization, grayscale, Gaussian filter, and segmentation. In this workflow only Dask APIs are used (`dask.array` and `dask.image`). The advantage of using those libraries is that they provide a high-level API and create the corresponding Dask task graph under the hood. We run one task graph per step and use the Breast Cancer Semantic Segmentation dataset [23].
Fine-tuned `ResNet152` batch prediction: We have fine-tuned the pretrained Pytorch ResNet152 image classification model on the supervised part of the Imagewang.[7] The dataset corresponds to a subset of 20 classes from the original ImageNet dataset. In this workflow we have three main functions decorated with `@dask.delayed` to create tasks: load, transform, and predict.
Training a regression model using `XGBOOST`: This workflow trains a regression model to predict trip duration using New York City High Volume For-Hire Vehicle trip records.[8] We have used the parquet data records from 2019 through 2024, with a total size of 20 GiB. High-level methods such as `xgboost.dask.train` and `xgboost.dask.predict` are used, and the underlying task graph is created automatically, thanks to the use of Dask libraries such as `dask.array` and `dask.dataframe`.

[5]https://examples.dask.org/

[6]https://saturncloud.io/docs/examples/python/pytorch/qs-03-pytorch-gpu-dask-single-model/

[7]https://github.com/fastai/imagenette

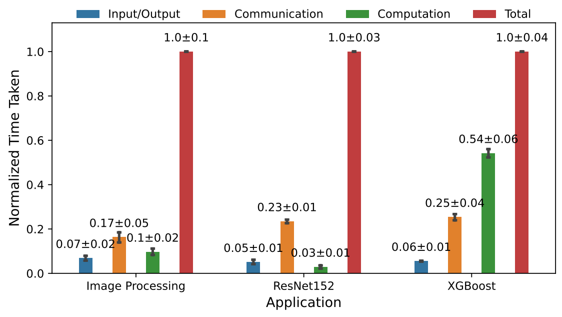

Fig. 3: Relative time spent per workflow in I/O, communication, and computation and total wall time.

C. High-level analysis

Figure 3 shows the relative time spent per workflow (Section IV-B), and Table I provides a high-level synopsis of the characteristics of these workflows. This analysis studies the variability observed in each of the three phases—I/O, communication, and computation—and the total wall time. The x-axis of Figure 3 shows the workflows, and the y-axis displays the normalized average time spent in each phase (we normalize the y-axis for readability as workflows vary in total duration). The error bars, a key element, depict variability across all runs of each workflow.

The I/O bar represents the sum of the I/O operations collected from Darshan reports, the communication bar is the sum of all incoming communications to the workers, and the computation bar is the sum of the computation time within tasks. The total bar represents the wall time for the workflow as a whole, including workflow coordination time (e.g., connecting to the scheduler, waiting for workers, creating the task graph) in addition to time spent in I/O, communication, and computation. Note that the I/O, communication, and computation times are non-exclusive and may overlap, and they can be different from run to run (see Table I). The overall `ImageProcessing` and `ResNet152` workflow wall times are relatively short (around one hundred seconds) and do not allow sufficient time to amortize the coordination overheads, leading to a disproportionately long total time. This is not the case for `XGBOOST`. In this figure we gather data from multiple runs and multiple sources within each run and present an overview of each phase's duration and variability.

[8]https://www.nyc.gov/site/tlc/about/tlc-trip-record-data.page

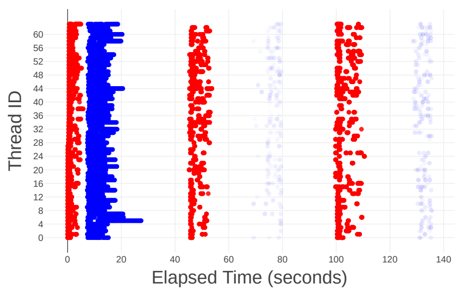

Fig. 4: Per-thread I/O of `ImageProcessing` workflow over time. Red and blue are read and write operations, respectively, and transparency corresponds to I/O size.

However, this gives only a very high-level summary about the performance. We delve into more details of each phase in the next section in order to understand the performance of individual tasks and workflows.

Workflows	`ImageProcessing`	`ResNet152`	`XGBOOST`
Task graphs	3	1	74
Distinct tasks	5440	8645	10348
Distinct files	151	3929	61
I/O operation	5274–5287	2057–2302[9]	867–1670
Communications	3141–3247	3751–3976	1464–2027

TABLE I: Workflow Characteristics.

D. *Detailed analysis*

In this section we show the most relevant analysis for each workflow. These analyses were not possible with the default Dask dashboard and give important insights about I/O behavior, communication patterns, task duration, warning distribution, and all types of correlations between them. This is a small subset of the type of analysis we are able to perform but do not show here; examples include task category (type) analysis within one or multiple runs (performance, variability, distribution, I/O per task, and so), zooming through a specific time period (get all events, compute/communication/I/O statistics), and comparison of scheduling strategies over runs such as whether tasks were scheduled in the same order or not.

1) *I/O distribution within a run*

Figure 4 presents the I/O characteristics of the `ImageProcessing` workflow across threads, as the workflow progresses. The x-axis shows the application's elapsed time, the y-axis shows the thread ID, horizontal lines indicate I/O duration, the color represents the type of the I/O (read in red, write in blue), and the opacity of the lines represents relative I/O size—the darker the color, the larger the I/O size.

We observe three read phases (red), each followed by a write phase. The written images in Phases 2 and 3 are smaller (a few kilobytes, hence appearing as a lighter blue) than the original images (80 MB). We observe small I/O sizes compared with the image sizes: 10–25 read operations of 4 MB each are performed per image. This translates to 10–25 I/O per Dask task calling `dask_image.imread`, which may be a factor contributing to performance variability that we were not able to see at the Dask level.

[9]The I/O operation count for `ResNet152` is incomplete due to default Darshan instrumentation buffer limits. We will increase this limit and explore the impact in future work.

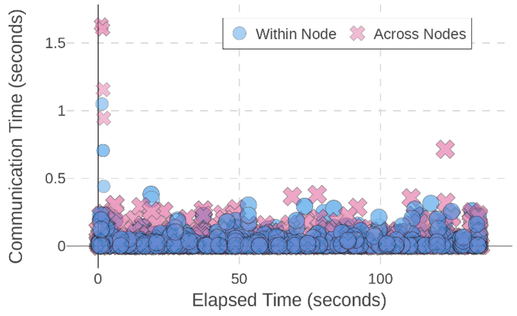

Fig. 5: Time spent in interworker communication over time for `ResNet152`. The color and shape represent whether the workers were on the same node, and the relative size corresponds to the transfer size, respectively.

As noted in Table I, the former workflow comprises three distinct task graphs while the latter is a single task graph. The three `ImageProcessing` task graphs are executed in sequence, which means that transitions between task graphs act as a form of synchronization, in turn producing bursts of simultaneous I/O activity rather than a uniformly distributed pattern. This is likely to make the `ImageProcessing` workflow more sensitive to fluctuations in storage system performance at scale and thus hinder performance reproducibility.

2) *Communication distribution within a run*

Figure 5 illustrates the variability in communication duration as the size of messages varies. The x-axis shows the sizes of messages transferred by different threads of the `ResNet152` workflow, the y-axis shows the time spent in a communication (seconds), and the color indicates whether a communication is performed across nodes or within a single node. From the figure we can observe several communications near the beginning of the workflow where the communication time was relatively long, and these communications are almost evenly split between inter- and intranode. Further data is needed to explain why these small communications exhibit this performance abnormality.

3) *Task duration and Dask warnings*

This analysis aims to connect multiple variables to visualize their relationships and interactions across different dimensions, highlighting patterns and correlations in the data. The first column displays the workflow's elapsed time, the second shows the task category, the third indicates which thread performs the task, the fourth presents the task output size in megabytes, and the fifth column shows the overall task duration in seconds. We use a color scale from white to red to represent task durations; red lines indicate tasks with longer durations. Examples of task categories include `read_parquet-fused-assign`, `getitem`, `random_split_take`, and `drop_by_shallow_copy`.

From Figure 6 we observe that the longest tasks (red lines)

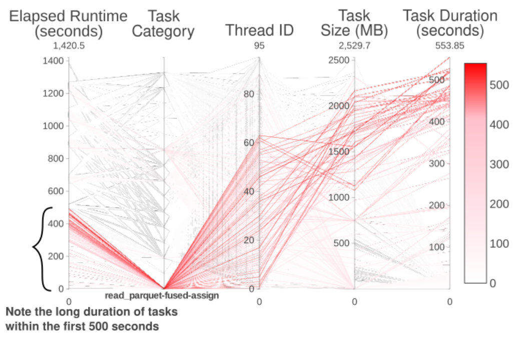

Fig. 6: Parallel coordinate chart of tasks in XGBOOST.

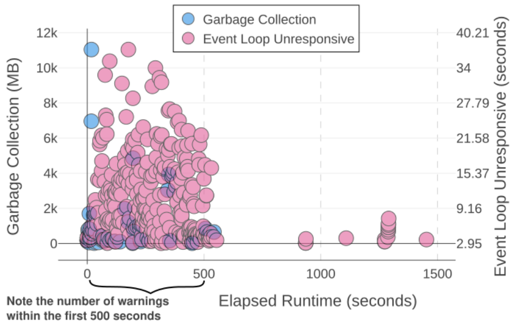

Fig. 7: Distribution of warnings in XGBOOST.

belong to the `read_parquet-fused-assign` category, which includes I/O and other operations. This task category arises from Dask's task-graph optimization process, where I/O operations are combined with consuming tasks into a single node of the task graph to enhance data locality. We also notice that the size of these tasks, which represents the size of the task's output data in Dask, is significantly larger than the recommended 128 MB by the Dask developers. This observation indicates a potential cause of suboptimal workflow performance, including performance variability. Note that tasks with sizes smaller than 128 MB have light grey lines leading to the bottom of the task duration coordinate.

We also collect warnings from the Dask scheduler and worker logs regarding the responsiveness of worker's event loop[10] and garbage collection events. We hypothesize that these warnings may be correlated with the slowdown of the Dask system and running tasks. Figure 7 depicts the distribution of these warnings. From this figure we indeed observe that there are 297 *unresponsive event loop* warnings generated in the first 500 seconds of the workflow, which correlates perfectly with the long-running `read_parquet-fused-assign` tasks (red lines).

E. Task provenance summary

Thanks to our multisource data collection, correlation, and analysis, we are able to construct a full lineage of every task in the workflow. Figure 8 shows a summary of the provenance data of a given task, whose key is `('getitem-_get_categories-24266c..', 63)`.

[10]Dask is built on Tornado and uses coroutines for concurrency.

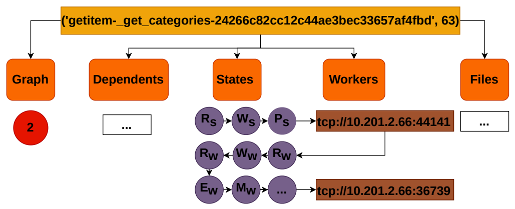

Fig. 8: Example of task provenance summary.

This task comes from the XGBOOST workflow; it has been submitted within the second task graph. We also have full dependency information, including status and location, but it is omitted from this figure because of space limitations. Every state transition is captured with the location and timestamp (example of a path under the *states* node in the figure). Once the task in computed, the output of the task lives in the distributed memory of Dask. We capture the data movements (tasks) between workers, and thus all locations of the given task, if duplicated. Moreover, we capture high-fidelity I/O records including the parallel file system (PFS), file path, I/O type, size, location in the file, and timestamp.

V. Lessons Learned

Performance and data provenance studies are more challenging with distributed task-based workflows than with classical MPI workflows, as tasks are dynamically dispatched to workers at runtime rather than statically assigned to known processes in advance. This dynamicity results in performance unpredictibility and variability even when running in the same environment and configuration. Dask uses various heuristics to optimize task placement and memory footprint, but factors such as allocated nodes, physical distance between the scheduler and the worker nodes, and network topology contribute to a high performance variability. Additionally, initial task placement can lead to different communication patterns (see number of communications in Table I), further impacting performance. Moreover, some workflow management systems implement work stealing strategies, which occur when idle workers request to run tasks originally dispatched to busy workers. Work stealing is a runtime decision that may negatively impact overall performance because of expensive data movements or unforeseen effects in future task dispatching.

Performance and data provenance study is nontrivial. Hidden details from the user perspective (e.g., I/Os, communications, work stealing) render data gathered from a single source insufficient to fully understand performance and its variability. Therefore, data from different layers in the software stack is collected by using various tools to provide complementary insights into task-based workflow performance.

Furthermore, aggregating and correlating diverse data from different sources into a single database and ensuring FAIR principles pose challenges. Even when the data collected at each layer respects the FAIR principles within the tool/layer, the aggregated data might not. For instance, without adding the POSIX thread ID and timestamps to both Darshan and Dask reports, I/O operations would not be correlated with the corresponding tasks, thus creating a lack of interoperability. Even if we do not have a perfect FAIR system yet, we have stored the data and metadata in a unique tabular format, with at least one common identifier between every two different data sources. For instance, tasks are identified by unique keys generated by Dask, start/end timestamps, the worker where they ran, and POSIX thread IDs. The scheduler and workers are identified by their IP/Port addresses and hostnames. I/O operations are identified by hostnames, POSIX thread IDs, and timestamps. Depending on the information needed, one or multiple identification fields can be utilized, and data coming from different sources is findable with the same identifiers.

VI. Conclusions and Future Work

Performance characterization of HPC workflows is challenging and complex for distributed task-based workflows, where low-level aspects may be hidden and there is no prior knowledge of resource management or task scheduling. In this paper we proposed a layered architecture to collect performance and data provenance of HPC workflows. We have provided an implementation using Dask.distributed WMS, alongside the proven I/O characterization tool Darshan and the Mofka streaming service. We have showed a subset of the analysis we can perform using data collected from multiple layers, and we have shared our lessons learned on data collection and aggregation, as well as performance characterization insights for task-based workflows.

In future work we will run larger-scale studies and explore collecting data from other sources such as TAU [10] and system-level network monitoring tools. We will shift to capturing Darshan records and pushing them to Mofka at runtime to have a fully online system. We also will explore options for dynamically adjusting our data capture in response to changes in workflow behavior. Although anticipated to be negligible, future work will include a thorough performance characterization of the overhead of Darshan and Mofka within Dask workflows. Another direction to consider is how to better understand distributed workflows running on heterogeneous architectures such as GPUs and FPGAs. Collecting data from tools such as NVIDIA NSIGHT and PyTorch Profiler for artificial intelligence workloads would improve our understanding of these workflows.

Acknowledgments

This material is based upon work supported by the U.S. Department of Energy (DOE), Office of Science, Office of Advanced Scientific Computing Research, under grant 0000269227 and contracts DEAC02-06CH11357 and DESC0012704.

References

[1] F. da Silva *et al.*, "A community roadmap for scientific workflows research and development," in *2021 IEEE Workshop on Workflows in Support of Large-Scale Science (WORKS)*, 2021, pp. 81–90.

[2] Y. Babuji *et al.*, "Parsl: Pervasive parallel programming in Python," in *Proceedings of the 28th International Symposium on High-Performance Parallel and Distributed Computing*, ser. HPDC '19. New York, NY, USA: Association for Computing Machinery, 2019, pp. 25—-36.

[3] M. Rocklin, "Dask: Parallel computation with blocked algorithms and task scheduling," in *Proceedings of the 14th Python in Science Conference*, K. Huff and J. Bergstra, Eds., 2015, pp. 130–136.

[4] A. Merzky, M. Turilli, M. Titov, A. Al-Saadi, and S. Jha, "Design and performance characterization of RADICAL-Pilot on leadership-class platforms," 2021.

[5] N. A. of Sciences Engineering and M. (NASEM), *Reproducibility and replicability in science*. National Academies Press, 2019.

[6] M. D. Wilkinson *et al.*, "The FAIR guiding principles for scientific data management and stewardship," *Scientific Data*, vol. 3, no. 1, pp. 1–9, 2016.

[7] R. B. Ross, G. Amvrosiadis, P. H. Carns, C. D. Cranor, and M. Dorier, "Mochi: Composing data services for high-performance computing environments," *Journal of Computer Science and Technology*, vol. 35, pp. 121 – 144, 2020.

[8] S. Snyder, P. Carns, K. Harms, R. Ross, and G. K. Lockwood, "Modular HPC I/O characterization with Darshan," in *2016 5th Workshop on Extreme-Scale Programming Tools (ESPT)*, 2016, pp. 9–17.

[9] L. Adhianto *et al.*, "HPCTOOLKIT: Tools for performance analysis of optimized parallel programs http://hpctoolkit.org," *Concurr. Comput. : Pract. Exper.*, vol. 22, no. 6, pp. 685–701, April 2010.

[10] S. S. Shende and A. D. Malony, "The Tau parallel performance system," *Int. J. High Perform. Comput. Appl.*, vol. 20, no. 2, pp. 287—-311, may 2006.

[11] P. Moritz, R. Nishihara, S. Wang, A. Tumanov, and R. Liaw, "Ray: A distributed framework for emerging AI applications," in *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'18. USA: USENIX Association, 2018, pp. 561—-577.

[12] E. Deelman *et al.*, "Pegasus: A framework for mapping complex scientific workflows onto distributed systems," *Scientific Programming*, vol. 13, no. 3, pp. 219–237, 2005.

[13] A. Agelastos *et al.*, "The lightweight distributed metric service: A scalable infrastructure for continuous monitoring of large scale computing systems and applications," in *SC '14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2014, pp. 154–165.

[14] J. Kreps *et al.*, "Kafka: A distributed messaging system for log processing," in *Proceedings of the NetDB*, vol. 11, no. 2011. Athens, Greece, 2011, pp. 1–7.

[15] M. Dorier, G. Antoniu, R. Ross, D. Kimpe, and S. Ibrahim, "CALCioM: Mitigating I/O interference in HPC systems through cross-application coordination," in *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, 2014, pp. 155–164.

[16] P. Carns *et al.*, "Understanding and improving computational science storage access through continuous characterization," *ACM Trans. Storage*, vol. 7, no. 3, oct 2011.

[17] J. Luettgau, S. Snyder, T. Reddy, N. Awtrey, and K. Harms, "Enabling agile analysis of I/O performance data with PyDarshan," in *Proceedings of the SC '23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis*, ser. SC-W '23. New York, NY, USA: Association for Computing Machinery, 2023, pp. 1380—-1391.

[18] C. Xu *et al.*, "Dxt: Darshan extended tracing," Argonne National Lab.(ANL), Argonne, IL (United States), Tech. Rep., 2017.

[19] M. Leinhauser *et al.*, "Performance analysis of picongpu: particle-in-cell on GPUs using NVIDIA's NSight systems and NSight compute," Oak Ridge National Lab.(ORNL), Oak Ridge, TN (United States), Tech. Rep., 2021.

[20] D. Boehme *et al.*, "Caliper: performance introspection for HPC software stacks," in *SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2016, pp. 550–560.

[21] W. McKinney *et al.*, "Pandas-powerful Python data analysis toolkit," *Pandas—Powerful Python Data Analysis Toolkit*, vol. 1625, 2015.

[22] A. Gueroudji, "Distributed Task-Based In Situ Data Analytics for High-Performance Simulations," Theses, Université Grenoble Alpes [2020-....], May 2023.

[23] M. Amgad *et al.*, "Structured crowdsourcing enables convolutional segmentation of histology images," *Bioinformatics*, vol. 35, no. 18, pp. 3461–3467, 02 2019.