Expanding Community Access to Real-World HPC Application I/O Characterization Data Using Darshan

Shane Snyder
ssnyder@mcs.anl.gov
Argonne National Laboratory
Lemont, Illinois, USA

Philip Carns
carns@mcs.anl.gov
Argonne National Laboratory
Lemont, Illinois, USA

Kevin Harms
harms@alcf.anl.gov
Argonne National Laboratory
Lemont, Illinois, USA

Robert Latham
robl@mcs.anl.gov
Argonne National Laboratory
Lemont, Illinois, USA

Robert Ross
rross@mcs.anl.gov
Argonne National Laboratory
Lemont, Illinois, USA

Abstract

High-performance computing systems are deployed with massive, distributed storage subsystems to meet the demands of data-intensive applications. While these storage systems offer impressive peak performance, it is often only attainable in idealized scenarios not reflective of production workloads. In general, there continues to be a lack of community understanding of the I/O performance characteristics of real-world applications.

The Darshan tool can help characterize application I/O behavior, but unfortunately Darshan logs are generally kept private to protect sensitive user-identifying information. This drastically restricts the impact of Darshan data, precluding the ability to analyze I/O behavior in a broader context. Public exchange of Darshan data could increase community understanding and inspire novel systems research but requires established processes for publishing anonymized, full-resolution Darshan logs.

To help address this problem, we developed an automated workflow for anonymizing and publishing Darshan data captured on Polaris, an HPE Apollo system at the ALCF. This continuously updated dataset includes over 600,000 logs and is growing daily. In this paper we describe this new community resource, demonstrate its utility by highlighting platform-wide characteristics and outlier job behavior, and explore best practices for continuously capturing and publishing system telemetry data.

CCS Concepts

• **General and reference** → **Performance**; **Measurement**; • **Information systems** → *Distributed storage.*

Keywords

Darshan, I/O Characterization, Performance Monitoring, Storage Systems, High-Performance Computing, Workflow Automation

CUG '25, New York, NY

ACM ISBN 978-x-xxxx-xxxx-x/YYYY/MM
https://doi.org/XXXXXXX.XXXXXXX

ACM Reference Format:
Shane Snyder, Philip Carns, Kevin Harms, Robert Latham, and Robert Ross. 2025. Expanding Community Access to Real-World HPC Application I/O Characterization Data Using Darshan. In *Proceedings of Cray User Group (CUG '25).* ACM, New York, NY, USA, 13 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

Modern high-performance computing (HPC) systems rely on large-scale, distributed storage subsystems to satisfy the I/O demands of increasingly data-intensive applications. While these storage systems can deliver impressive top-line performance, it is often attainable only in idealized scenarios that do not reflect the I/O behavior of real applications running in production. In practice, application I/O performance can vary greatly depending on factors such as access patterns, application scale, interference from other jobs, and system configuration. Unfortunately, there continues to be a general lack of understanding within the HPC community regarding how these factors influence application I/O performance, due in large part to a need for concrete data characterizing the I/O behavior of diverse real-world applications.

The Darshan I/O characterization tool [12] can help address this problem by providing HPC users with insights into the I/O behavior of their jobs. However, users generally only have access to Darshan data generated by their own jobs and have no way to compare, contrast, and interpret this data in a broader context. This situation limits the overall impact of Darshan data to localized analysis of individual applications rather than enabling comprehensive, systemwide insights. Opening access to Darshan log data captured at HPC facilities could offer the I/O research community crucial data on real-world access patterns and performance characteristics, advancing the state of the art in storage system understanding and optimization. Achieving this goal, however, requires established processes for publishing Darshan logs in a manner that preserves their full-resolution characterization data while protecting sensitive, user-identifying information.

In this work we aim to address this gap in community I/O performance understanding by publishing anonymized Darshan logs collected from real-world applications running on the Polaris HPC system at the Argonne Leadership Computing Facility (ALCF). To this end, we develop an automated workflow for the collection, anonymization, and publication of Polaris Darshan logs, making

them accessible to the broader I/O research community. To demonstrate the utility of this dataset, we present a preliminary analysis that uncovers high-level trends and common performance inefficiencies in the I/O behavior of production workloads on Polaris. Additionally, we share our perspectives on best practices for capturing large-scale telemetry datasets and discuss how this data can be leveraged to improve application and system-level performance.

This paper is organized as follows. Section 2 provides background and summarizes other research related to this work. In Section 3 we discuss the implementation details for our Darshan log collection workflow on the ALCF Polaris system and provide general best practices for capturing system telemetry data in production environments. In Section 4 we perform a preliminary analysis of the Darshan log collection to offer insights into the general I/O behavior of production jobs from Polaris. Section 5 summarizes our conclusions and potential next steps for this work.

2 Background and Related Work

In this section we provide a brief overview of the Darshan I/O characterization tool and relevant details on the ALCF Polaris system, including its hardware configuration, available storage systems, and Darshan deployment. We also discuss related work in the context of this research.

2.1 Darshan I/O Characterization Tool

Darshan is an I/O characterization tool designed specifically for understanding the storage access patterns of HPC applications. It provides detailed statistics on each file accessed by an application, with this data captured comprehensively across the various layers of the HPC I/O stack, from low-level storage interfaces to high-level data management libraries. Darshan's design is centered around lightweight and transparent instrumentation methods that enable full-time characterization of production jobs at HPC facilities. It limits its potential application perturbations by using a bounded amount of memory for storing instrumentation data and defers the more costly process of aggregating and logging this data to application shutdown time. Additionally, Darshan offers a range of analysis tools and interfaces that can be used to inspect logs and interpret results. These include a Python-based framework called PyDarshan [8] that enables integration of Darshan data into popular Python data analysis packages such as `pandas` and `matplotlib`.

The data captured by Darshan at HPC facilities is commonly analyzed by users and system administrators to help better understand I/O workload characteristics of jobs and, ideally, to guide application performance tuning decisions. Beyond the first-order impact on users and system administrators, this Darshan data captured at HPC facilities could serve as a foundation for broader I/O community research activities, such as the following:

- Analyzing prevalent systemwide application I/O trends and performance characteristics
- Clustering job populations at a facility in terms of observed I/O characteristics
- Reconstructing representative application I/O workloads for use in storage system workload modeling

Because of its capture of sensitive user-identifying information, however, Darshan data gathered at facilities tends to be private and is visible only to the users that generated it or to system administrators. This situation generally limits Darshan's overall ability to inform broader research community investigations of I/O performance behavior on production HPC systems.

2.2 ALCF Polaris

Polaris is a 520-node HPE Apollo 6500 Gen 10+ based system deployed at the ALCF offering a peak performance of 44 petaflops. Each node has one 2.8 GHz AMD EPYC Milan 7543P 32-core CPU (with 512 GB of DDR4 RAM), four Nvidia A100 GPUs, and two Slingshot 11 network adapters. Polaris offers a few different storage systems intended for users' data:

- Eagle/Grand, Lustre file systems, each composed of 160 object storage targets (OSTs) and 40 metadata targets (MDTs), with a total capacity of 100 PiB and peak performance of 650 GiB/s
- home, a Lustre file system meant for storage of non-I/O intensive user data
- temporary node-local storage for jobs, using two SSDs (XFS formatted) with a total capacity of 3.2 TiB and peak performance of 6 GiB/s per node

Darshan software is installed and enabled by default on Polaris using Cray Programming Environment modules. The Darshan module integrates directly into the Cray build environment and automatically links Darshan libraries into application executables (both dynamically and statically linked). This approach enables transparent and seamless instrumentation of Polaris applications without requiring any action on the part of end users.

2.3 Related Work

A number of attempts have been made to anonymize and publish Darshan data captured on production HPC systems for sharing with the research community. For example, anonymized Darshan datasets were captured and published for previous-generation HPC systems at the ALCF and at the National Center for Supercomputing Applications, with this data serving as the basis for numerous research studies into HPC I/O behavior [2, 6, 9, 11]. While these datasets have been valuable to the HPC I/O research community, they quickly become outdated, as they represent snapshots in time and are rarely, if ever, updated. Also underway is an ongoing effort to maintain a searchable, collaborative I/O trace repository of Darshan data [10], with the community contributing Darshan logs (and additional application/system context) to a public repository that can be used to drive general HPC I/O research activities. This effort increases community accessibility to I/O characterization data from diverse applications, but it does not offer the systemwide perspective necessary for understanding overall storage behavior on HPC systems. Comprehensive, continuously updated I/O characterization data for modern HPC systems is desperately needed to bolster community understanding of rapidly evolving HPC technologies.

Other previous work [7] published in-depth datasets characterizing systemwide I/O behavior over a year's time on production systems at NERSC (Cori, Edison) and the ALCF (Mira). Published data includes raw Darshan log data and file system monitoring data, as well as software packages for combining and analyzing this

data. Compared with our work here, these datasets provide comprehensive systemwide I/O characterization from the application itself to the underlying file system. However, a key distinction is that our dataset captures I/O behavior from real-world application workloads, whereas the prior datasets relied on regularly executed synthetic benchmarks intended to assess system I/O performance trends.

Aside from Darshan, other tools have been used to help provide deeper insights into I/O behavior of HPC applications. For example, Recorder [14] is a detailed, multilevel I/O tracing tool intended for deeper analysis of the behavior of HPC applications. DFTracer [4] is another novel I/O instrumentation tool intended for use in HPC, specifically in the instrumentation of artificial intelligence/machine learning (AI/ML) workflows. While these tools offer appealing features for in-depth analysis of I/O behavior (e.g., in-depth tracing of multiple I/O interfaces or the application itself), they are used mostly for targeted analysis and are not widely deployed across production workloads. As a result, they are less suited for establishing comprehensive views of systemwide I/O behavior, which is a key strength of Darshan and the focus of our work here.

HPC facilities are also well positioned to publish telemetry datasets collected from their production systems. For example, the ALCF maintains an extensive public repository[1] providing datasets captured from job schedulers, RAS systems, and other sources on current and previous generation HPC systems. We rely on ALCF datasets in this work, correlating Darshan logs with detailed job scheduler data to better understand the extent of Darshan instrumentation coverage of Polaris jobs.

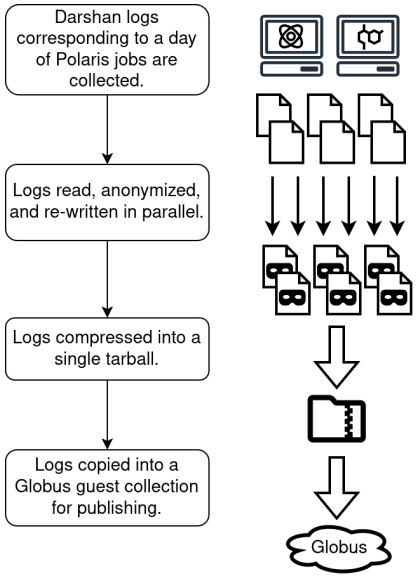
3 Establishing a Continuously Updated Community Dataset of Real-World HPC Application I/O Behavior

To help address the community need for comprehensive I/O characterization data that is representative of modern HPC applications and systems, we have developed a workflow for anonymizing and publishing Darshan data captured in production on the ALCF Polaris system. The goal of this work is not only to help fill an outstanding knowledge gap in the HPC I/O research community but also to establish general best practices for the continuous capture and publication of system telemetry data. Ultimately, we envision that this work can serve as a blueprint for broader collection and publication of the vast amounts of telemetry data being captured on modern HPC systems. Opening up access to this telemetry data can help maximize its impact beyond facility-centric operational analysis, enabling collaborative, community-driven research efforts that can yield new insights into system behavior and application performance.

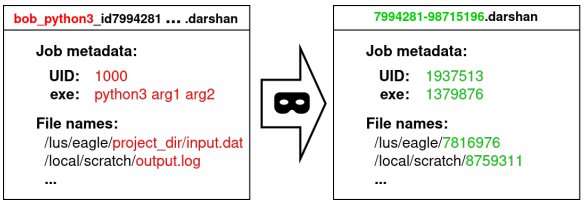
3.1 ALCF Polaris Darshan Log Collection

As part of this work, we have implemented a workflow for publishing anonymized Darshan logs collected from production jobs running on Polaris. A general overview of this workflow is provided in Figure 1 and described in more detail below.

[1]https://reports.alcf.anl.gov/data/index.html



(a) Diagram depicting the overall steps of this workflow and the transformation of Darshan data.



(b) The anonymization step of the workflow is used to mask sensitive user-identifying data contained in Darshan logs.

Figure 1: Overview of the workflow for anonymizing and publishing Darshan logs collected on the ALCF Polaris system.

As mentioned previously, Darshan transparently integrates into Polaris applications at build time via the Cray Programming Environment. When instrumented applications terminate, the Darshan library writes the captured characterization data to a log file stored in a centralized directory. To simplify the process of finding logs in this central directory, it is organized into a year, month, and day subdirectory hierarchy based on the date the application terminates. Further, each log file name has helpful identifying information, including the corresponding username, application name, and job ID. Although the Darshan log directory is shared by all users, log data is accessible only by the users that generated them (and by system administrators).

The workflow begins by collecting a full day (specifically, the oldest day not yet processed by the workflow) of Darshan logs from this central repository and anonymizing sensitive user-identifying details in them. This sensitive information includes the Darshan log file names themselves, as well as job metadata captured in the log (UIDs and command line info) and file names accessed by the application. For the log file names, this workflow simply

removes user-identifying details and produces a randomized output log name prefixed with the original job ID. Preserving the job ID allows for job-aware analysis, as we will demonstrate in Section 4.1. For the data contained within the log, a hash function is used to transform the sensitive information into a string of numbers. This hash function uses the same seed value throughout the entire log collection to enable consistent mapping of these values for subsequent log analysis (e.g., for grouping logs based on UID). For file names accessed by the application, only the portion of the file name following the mount point prefix is anonymized. This enables clear identification of which storage system each file was originally located, without exposing potential user-identifying information embedded in the remainder of the file name.

The anonymization process itself requires reading (and then decompressing) Darshan log data, modifying the user-identifying portions of it, and rewriting it (after recompressing it) to a new log file. Therefore, a naive serial implementation of this process can be a serious bottleneck to this workflow, as tens of thousands of Darshan logs can be captured on Polaris in a given day. To address this, we embrace the embarrassingly parallel nature of this workflow stage and use multiple processes to independently anonymize distinct subsets of the Darshan log dataset. This strategy is straightforward to implement in our anonymization script, which is written using Python, by using `ThreadPoolExecutor` (from the `concurrent.futures` module). Thread pools are chosen rather than process pools (i.e., `ThreadPoolExecutor`) because most of the anonymization is actually carried out via a Darshan command line tool (`darshan-convert`) that is issued using the Python `subprocess` module. While Python is not traditionally used for performance-critical computing, this approach is more than adequate for the demands of this workflow. In practice, an entire day's worth of logs (often numbering in the thousands) can typically be processed in well under a minute, making runtime a non-issue.

After anonymization has completed, the entire log set is compressed into a single tarball of logs corresponding to that particular day. Aside from reducing the size of the log collection, a key benefit of archiving the logs into per-day tarballs is to limit file system stress when copying and sharing the dataset, given the massive amount of logs that could be captured in a single day. Table 1 provides some overall compression statistics for the entirety of the 2024 portion of this dataset, showing that the final collection of compressed tarballs results in a 55% size reduction compared with the original Darshan logs stored on Polaris. Notably, the majority of the compression gains (52%) are due to our workflow recompressing log data in `bzip2` format (instead of Darshan's native runtime compression using `zlib`), with this step performed implicitly as part of the anonymization process described above.

The final log tarball for that day is then copied to a directory for anonymized Darshan log data maintained on the Eagle file system mounted on Polaris. This directory is made publicly available using Globus *guest collections*, which allow external access and enable transfers of this data to other Globus-enabled endpoints (e.g., on Polaris, on other systems, on personal laptops). This guest collection mirrors the year, month, and day directory hierarchy employed in the original non-anonmyized Darshan log directory, with the day directories simply containing the final tarball from this workflow that can be unpacked to produce the anonymized Darshan logs. This log collection will be continuously be updated using the workflow described here. **As a key contribution of this work, we release the full Polaris Darshan log collection [13] as described above for broader community use.** We also make available the full set of scripts used to collect and anonymize this dataset, enabling reproducibility and facilitating adoption of similar workflows on other HPC systems.[2]

Log Compression Method	Total Size (GiB)	Size Reduction (% from original)
`zlib`-compressed (original)	10.82	–
`bzip2`-compressed	5.20	52%
`bzip2`-compressed tarball	4.83	55%

Table 1: Comparison of Darshan log compression methods using entire ALCF Polaris log collection dataset from 2024.

This log collection workflow leverages ALCF CI resources developed and deployed as part of the Exascale Computing Project [1], namely, ALCF's internal GitLab CI instance and the Jacamar CI driver. Specifically, the workflow is implemented as a GitLab pipeline that is scheduled to run nightly on Polaris. A shell script is used to determine which days require processing and to invoke the log collection workflow on portions of the central Darshan log directory accordingly. This process is fully automated and is resilient to system downtime, enabling a truly "set-and-forget" approach to maintaining the log collection dataset.

We additionally note that users may still choose to opt out of the anonymized publication of their Darshan logs. Darshan provides mechanisms for users to redirect log output to private directories rather than a shared facility-wide location. This capability allows users to take advantage of Darshan's instrumentation for their own analysis while remaining excluded from facility-wide data collection.

3.2 Best Practices for Capturing and Publishing System Telemetry Data

Based on our experiences with the Polaris Darshan log collection, we offer some general best practices to the community on the continuous capture and publication of large-scale system telemetry datasets.

Carefully consider appropriate dataset transformations. The process of collecting telemetry data for publication presents a natural opportunity to apply appropriate transformations to the data to increase its usability. These transformations include the following:

- Scrubbing of data to sanitize it in various ways (e.g., normalization, anonymization, error correction)
- Reorganization of data to simplify subsequent analysis tasks
- Compression of data to minimize storage overheads in ways that may not be feasible at runtime
- Annotation of data with additional context to make it easier to interpret

[2] https://github.com/darshan-hpc/polaris-log-collection-scripts-cug25/tree/main/log-collection-workflow

These are critical because processes for capturing and storing telemetry do not typically account for these tasks, instead preferring to use simple, low-overhead methods for persisting data.

The primary transformation for the Darshan log collection workflow involves anonymization of user-identifying data contained within the logs. Additionally, as described in the preceding section, the workflow applies more efficient compression techniques that dramatically reduce the storage requirements for the log collection and a reorganization of data into tarballs to minimize transfer overheads. Overall, these transformations optimize storage and sharing of the log collection while preserving privacy of Polaris users.

Leverage continuous integration (CI) frameworks for automated management of datasets. While the curation and management of datasets are not strictly CI in definitional terms, infrastructure and features provided by CI frameworks are generally appealing for automating this sort of task. Examples include the following:

- CI build agents (e.g., GitLab runners) allow for the deployment of data collection workflows directly on the system capturing telemetry data.
- A variety of CI trigger mechanisms can be used to execute data collection workflows in different scenarios, including manual triggers for on-demand collection and scheduled triggers for continuous collection.
- CI dashboards provide enhanced logging and visibility capabilities that enable tracking and reporting the status of data collection workflows over time.

Collectively, the features provided by CI frameworks naturally support the automation and management of data collection workflows, reducing manual effort and improving reliability.

In our case we found GitLab CI pipelines greatly simplified the management of the Darshan log collection workflow. The ALCF's GitLab CI instance includes Polaris runners for both login and compute nodes, with our workflow using compute node runners to take advantage of more on-node resources and to limit impact on other users. Pipeline schedules allow for daily execution of the workflow, ensuring that logs from the previous day are anonymized and published. Additionally, pipeline status emails provide automated failure notifications and detailed pipeline logs facilitate rapid identification of underlying issues.

Gracefully detect and respond to workflow errors. Unexpected errors are inevitable when collecting and processing large-scale telemetry datasets. Gracefully handling encountered errors is a challenging problem, particularly in automated environments such as CI frameworks, where properly detecting and responding to diverse errors within complex workflows can be difficult. These errors can originate from the CI framework, from underlying data collection utilities, or from the system itself. While some errors can be ignored to allow the data collection and analysis process to continue, others may necessitate aborting this process entirely. In either case, errors need to be detected as they occur, addressed within the workflow, and ultimately reported so that failure scenarios can be better understood in the future. Striking the right balance between effective error handling and continuity of telemetry capture is key. In general, we recommend applying stricter error handling techniques during early workflow development to gain a deeper understanding of potential failure modes and to prevent the workflow from silently mishandling errors.

In the Darshan log collection workflow, we use strict error handling mechanisms in our scripts (e.g., `set -e` and `set -o pipefail` Bash options) by default. This approach is especially helpful for identifying unexpected errors as they occur and aborting the workflow, rather than silently failing and continuing execution. As an example, our initial workflow silently skipped log collection in cases where tens of thousands of logs were analyzed at once, because of an unexpected command line limit on the total number of files that can be passed to the `tar` command. Aborting on any error by default ensures that all failure modes are understood and are properly addressed in the workflow going forward. If an error is determined to not be critical, we modify our scripts to loosen error handling and log them as warnings instead. As another example, it is possible for Darshan logs to be corrupted (e.g., due to application memory issues, Darshan bugs, or file system failures), making them unreadable by tools used in our workflow. Rather than aborting in such scenarios, we log the details and report problematic logs at the end of the workflow so they can be further investigated.

Continuously analyze telemetry for broader insights. Aside from the primary goal of collecting and publishing large-scale system telemetry data, developed workflows can be used to obtain broader insights, for example into the behavior of the system being monitored or into the tool itself. Continuous capture of this sort of information can allow for tracking of behavioral changes over time and provide early indicators for system or application inefficiencies, evolving usage patterns, and so on. This information can be particularly useful to relevant stakeholders, including facility staff and the developers of telemetry capture tools, and can guide system optimization or tool improvements.

We envision analyzing regularly collected Darshan data to gain a better understanding of its production usage and to identify applications with I/O-intensive patterns that could potentially benefit from optimization. This capability, although not yet implemented, can help uncover instrumentation blind spots or general software bugs and inform subsequent tool improvements. Characterizing the behavior of the most I/O-intensive jobs that disproportionately impact overall system usage can lead to high-impact performance optimizations and improved system efficiency. We discuss these possibilities in more detail throughout Section 4.

Use established community tools for preserving and sharing datasets. One of the most critical decisions when publishing telemetry datasets is determining how to best preserve this data and share with the broader community. This step is key to ensuring that the community can easily find, download, and cite the dataset in their own research studies. Researchers have many available options in terms of data repositories that can be used for hosting their datasets and other research artifacts. Options range from completely generic open repositories such as Zenodo to community-specific repositories that typically have stricter review and curation policies. The best approach for sharing and persisting research datasets can vary and largely depends on the particular use case.

We use both Globus and Zenodo to manage different aspects of the Polaris Darshan log collection. Globus allows general public access to the Darshan logs via guest collections managed by the

ALCF. On the other hand, Zenodo is primarily used to provide a persistent record of the dataset maintained in Globus, allowing it to be cited in other works. The log collection documentation serves as the primary artifact associated with the Zenodo repository, with this documentation linking directly to the corresponding Globus guest collection containing the dataset. We found this setup to be most effective in our case, enabling the flexibility to continuously update the log collection dataset while still maintaining the ability to persistently reference it.

4 Polaris Darshan Log Collection Analysis

While HPC facilities that capture Darshan data are theoretically well positioned to perform more direct analyses of application I/O behavior, in practice, staff may lack the time, personnel, or specialized expertise to do so. By making anonymized logs publicly available, we enable the broader HPC I/O research community, which often possesses deep expertise in performance analysis and tool development, to contribute meaningfully to this effort. This more open model of collaboration can yield significant long-term benefits: the community builds tools and methods for identifying inefficiencies and characterizing I/O behavior which facilities can then incorporate into their operational workflows. The result is a virtuous cycle in which shared telemetry data fuels external research, ultimately feeding back to facilities to improve system and application performance. To demonstrate the value of this approach, we perform a preliminary analysis of the I/O characteristics of Polaris jobs using the anonymized public dataset, focusing on uncovering general systemwide I/O trends and inefficiencies in data-intensive jobs.

To support reproducibility and encourage broader community engagement, we have released the tools we developed and used for analyzing this massive dataset in an open-source GitHub repository.[3] These tools rely heavily on the PyDarshan framework [8], a Python package that simplifies extraction of Darshan log data in Pythonic format so that it can be analyzed using common packages such as `pandas` and `Matplotlib`. This repository will serve both as a reference for how to operate on the dataset and as a foundation for performing similar analyses on Darshan logs captured on other systems. By sharing a common set of tools and workflows, we hope to enable a consistent and extensible approach to I/O analysis across HPC systems.

4.1 Darshan Coverage of Polaris Jobs

We note that the Darshan logs collected on Polaris do not capture complete instrumentation data for all jobs executed on the system. Darshan may fail to log data from production jobs for a number of reasons:

- Applications can choose to unload the Darshan module when building their applications, preventing Darshan instrumentation at runtime.
- MPI applications with Darshan instrumentation enabled could crash or otherwise not call `MPI_Finalize()`, preventing Darshan from generating a log file.
- Applications not using MPI do not generate Darshan logs by default and rely on user opt-in by setting explicit environment variables. This limits Darshan insights into general serial applications and also many emerging Python-based frameworks (unless they leverage MPI-based backends such as `mpi4py` and `Horovod` + MPI).

Furthermore, Darshan does not provide any insight into I/O activities that occur outside the context of batch-scheduled jobs running on compute nodes. These include data management tasks executed on login nodes, as well as wide area and local transfers executed on ALCF data transfer nodes (DTNs), for example, using Globus. Generally speaking, this Darshan log collection dataset provides insight into the I/O behavior of only a subset of applications (and broader I/O activities) that run on Polaris.

To better quantify the extent of Darshan instrumentation of Polaris jobs, we define a metric called *coverage*. At the most basic level, coverage simply represents the fraction of system job activity (measured in node-hours) for which Darshan was able provide instrumentation data. This coverage value can be calculated by cross-referencing Darshan log metadata with PBS scheduler data captured and published independently by the ALCF.[4] Darshan's coverage over a given set of jobs can be expressed by using the following formula:

$$coverage = \frac{\Sigma\, node\_hours_{Darshan}}{\Sigma\, node\_hours_{PBS}}.$$

Accurately quantifying coverage can be challenging, however, because of misalignments in job scheduler accounting and the job metadata captured in Darshan logs. Specifically, job scheduler data can only indicate the start and end time of entire batch-scheduled jobs, which are often specified as scripts composed of potentially multiple job steps (i.e., calls to `mpiexec`) and other auxiliary commands (e.g., copying of files using `cp` or setting up experiment directories using `mkdir`). On the other hand, Darshan typically captures logs only for individual job steps corresponding to MPI applications, which may represent just a fraction of the job time accounted for by PBS scheduler logs. Additionally, PBS job logs provide information on total nodes used by an entire job, while Darshan collects information on the total number of processes an instrumented application used. This process essentially precludes the ability to calculate consistent job activity estimates using PBS job data and Darshan log data.

While the process of reconciling Darshan log data with less granular job scheduling data provided by PBS is inexact, we can still make reasonable approximations of Darshan coverage by making some simplifying assumptions. Given that it is not possible to calculate exact job node-hours using Darshan data, we opt to calculate the node-hours accounted for by Darshan jobs using data from PBS scheduler logs (which directly calculate total nodes and job time). To do so, we calculate the unique set of PBS job IDs accounted for in the set of Darshan logs and cross-reference these job IDs with the PBS job dataset. For each unique PBS job ID captured by Darshan, we simply assume that Darshan provides full coverage of that job's node-hours (as measured by PBS). While this simplification leads to an inflated Darshan coverage, it likely leads to a more reasonable

[3] https://github.com/darshan-hpc/polaris-log-collection-scripts-cug25/tree/main/log-collection-analysis

[4] https://reports.alcf.anl.gov/data/polaris.html

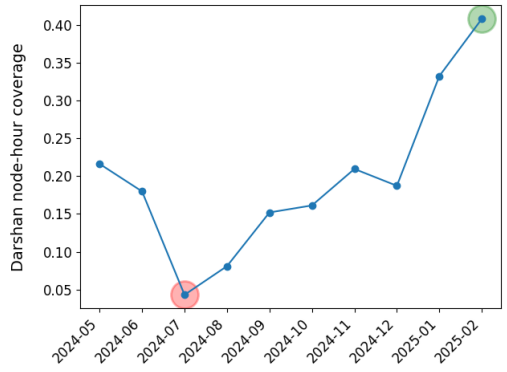


Figure 2: Darshan job coverage in terms of node-hours for 10 full months of the ALCF Polaris log collection. We highlight July 2024 as a month with abnormally low coverage and February 2025 as a month with abnormally high coverage and analyze these months in further detail in Figure 3.

estimate than would an alternative approach that scales node-hours by the amount of wall time observed in Darshan logs – this sort of approach would decrease Darshan coverage on the basis of job activity completely unrelated to I/O, including auxiliary commands and job scheduler overheads.

To estimate Darshan's job coverage on the Polaris system, we analyzed 10 full months of Darshan logs from May 2024 through February 2025 (totaling 611,606 logs). The results of this coverage analysis are provided for each month in Figure 2, with the aggregate coverage factor plotted for each month. In total, Darshan achieves a roughly 20% average monthly coverage on the basis of node-hours. This coverage value leaves obvious room for improvement for the Darshan deployment on Polaris, but it is relatively in line with prior evaluations of this metric [2, 3].

The results also show how dramatically Darshan's coverage can vary from month to month. As two extreme examples highlighted in these results, Darshan's coverage fell to under 5% in July 2024, while in February 2025 it rose to over 40%. To attempt to explain these anomalous results, we provide a more detailed analysis of Polaris project allocations in each of these months in Figure 3. The results for July 2024 in Figure 3a indicate that the top 20 projects in terms of node-hour usage (accounting for 90% of the total) almost exclusively did not use Darshan instrumentation. In particular, one project accounted for over 40% usage of node-hours this month and generated a minuscule amount of corresponding Darshan data. In contrast, the results for February 2025 in Figure 3b demonstrate totally different coverage characteristics for top projects, with Darshan providing almost full coverage for many of the top projects in terms of node-hour usage. These results highlight how susceptible Darshan's coverage factor is to high-impact projects that could dominate node-hour usage over given timeframes and that may or may not use Darshan instrumentation for varying reasons. To help address this situation going forward, we provide some discussion of potential techniques for generally improving and stabilizing coverage of Darshan deployments in Section 4.4.

4.2 Polaris Job Characteristics

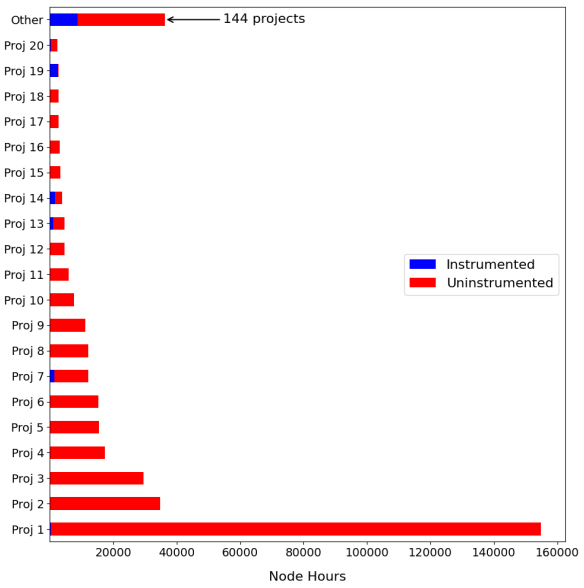
We start our analysis of the Polaris Darshan log collection by focusing on general I/O behavior and performance characteristics of the 10-month job population introduced in the preceding section. Note that our usage of the term *job* in the following discussions relates to individual application executions as captured by Darshan (i.e., calls to `mpiexec`), as opposed to broader batch-scheduled jobs using PBS (which could be composed of many independent application execution steps). Also, keep in mind that many jobs use multiple interfaces. Rather than aggregate statistics from each interface into a single value for each job, we produce multiple data points (i.e., one per interface) so we can better understand usage of each interface.

Table 2 provides a high-level overview of job I/O activity across the entire dataset, focusing on the total number of bytes and files accessed using different I/O interfaces. One immediate observation is that the total data transferred via the STDIO interface is significantly larger than that of both the POSIX and MPI-IO interfaces. Conventional wisdom in the HPC community holds that POSIX is the preferred low-level I/O interface for data-intensive applications, offering greater control over performance and direct compatibility with high-level I/O libraries and parallel file systems. These results indicate that, at least in aggregate, applications in this dataset do not necessarily follow this convention, which is a surprising result. Another observation is that MPI-IO usage shows a much lower number of total jobs and files but has a comparable total number of bytes accessed compared with POSIX. While this disparity may be larger than expected, it is intuitive that MPI-IO is primarily used by a smaller number of more data-intensive jobs. Another interesting result not specifically captured in the table is the fact that of the 611,606 jobs instrumented by Darshan over this timeframe, 205,448 (around 34% of total) did not access any files using POSIX, STDIO, or MPI-IO.

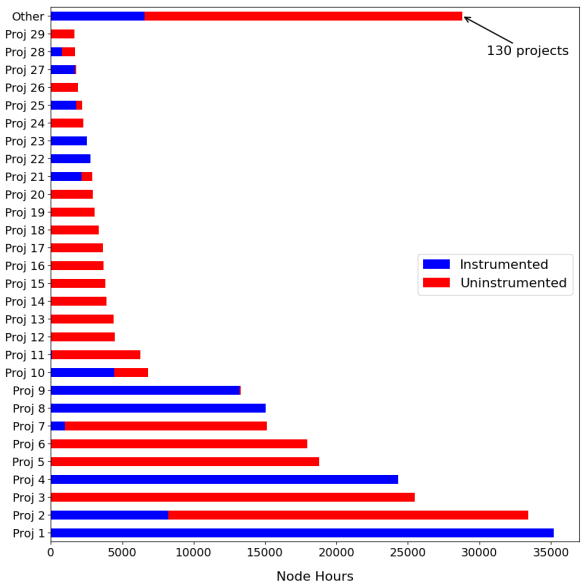
API	Total Jobs	Total Bytes (PiB)	Total Files
POSIX	195,512	1.53	20,221,863
STDIO	404,604	36.66	16,147,980
MPI-IO	18,216	1.42	523,071

Table 2: Polaris job I/O activity overview for various APIs.

To better indicate the distribution of different I/O characteristics across jobs in the dataset, we provided CDF plots of these metrics for each I/O interface in Figure 4. We can see that for the total bytes accessed by each job (Figure 4a), jobs using POSIX tend to access more data than jobs using STDIO, and MPI-IO jobs tend to access more data than jobs using POSIX. For total files accessed by each job (Figure 4b), we can see that POSIX jobs generally access the most files, followed by STDIO jobs and then MPI-IO jobs. In Figure 4c we additionally analyze sustained I/O performance distributions for all jobs. These results indicate that POSIX and MPI-IO distributions show similar performance trends across jobs, while STDIO distribution lags behind in performance for the majority of jobs. Overall, these results largely align with expected API usage patterns, although we again observe unexpected long tail behavior for STDIO jobs, especially in the bytes accessed CDF.



(a) July 2024 coverage by project: 20 projects account for 90% of coverage of all node-hours, and 144 projects (labeled Other) account for 10% of coverage.



(b) February 2025 coverage by project: 29 projects account for 90% of coverage of all node-hours, and 130 projects (labeled Other) account for 10% of coverage.

Figure 3: Darshan job coverage for projects from anomalous months, indicating instrumented (i.e., covered) and uninstrumented node-hours.

To attempt to quantify the heavy tail behavior for the total bytes accessed CDFs, we analyzed the top 1% of jobs for each I/O interface. We found that STDIO jobs in particular are extremely skewed, with 1% of jobs accounting for over 99% of all STDIO data. Conversely, we found that POSIX and MPI-IO jobs are similarly skewed but much less so than STDIO, with the top 1% of jobs responsible for 85% of POSIX data and 87% of MPI-IO data. These findings suggest that the large discrepancy in STDIO usage seen in Table 2 is likely driven by a small subset of jobs with abnormally high I/O activity. We investigate this phenomenon further in Section 4.3.

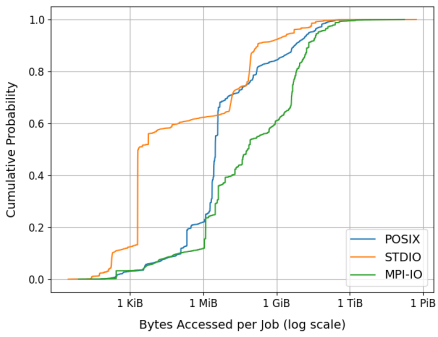
In Figure 5 we provide more details on the overall usage of Polaris storage systems using different I/O interfaces. Storage system utilization is given in terms of number of jobs in Figure 5a and in terms of total bytes accessed in Figure 5b. This data indicates that the Polaris home file system is accessed by the most jobs across all I/O interfaces, while the Eagle/Grand Lustre file systems serve the majority of data to jobs overall. This is not an unexpected result as applications are strongly encouraged to use the high-performance Lustre file systems for their data-intensive files. On the other hand, many applications are likely to access at least some files on the home file system, whether it be smaller application inputs/outputs, configuration files, or so on. However, there is a surprising amount of MPI-IO usage of the home file system. MPI-IO is typically used for scalable access to larger files, which is not an access pattern that is intended for the home file system, suggesting this could potentially be as a result of misconfigured file paths.

Notably, local scratch storage on Polaris compute nodes is used only by a small fraction of jobs. For example, just 1.5% of POSIX jobs and less than 1% of STDIO jobs access local scratch storage. MPI-IO usage of local scratch is understandably low, as these local storage devices are not visible across nodes, limiting their applicability in most parallel I/O use cases. Although used sparingly in the jobs analyzed in this work, local scratch storage devices offer appealing characteristics that could optimize the I/O performance of some Polaris applications. Specifically, these devices are great candidates for locally caching frequently accessed data, assuming the data is not too large (no more than a few TiB) and not collectively modified by processes spanning multiple compute nodes. Local copies of this data can limit high-latency interactions with remote storage systems such as Lustre and can avoid contention with application processes on other compute nodes.

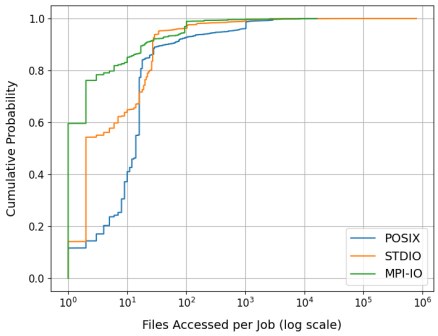
4.3 Case Studies of Inefficient I/O Behavior

In this section we examine a couple of Polaris jobs exhibiting extreme examples of inefficient I/O patterns. These workloads highlight significant performance bottlenecks and represent ideal candidates for optimization.

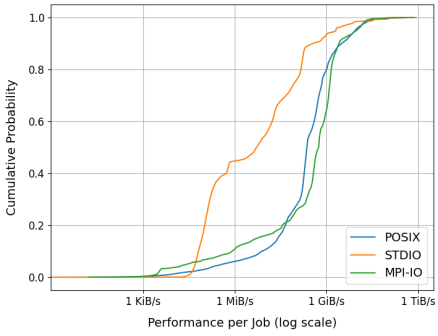
4.3.1 Case Study 1: MPI-IO collectives with misconfigured Lustre striping. The job in this case study was identified by ranking all MPI-IO jobs by their achieved I/O performance and manually examining the lowest-performing ones. This job was specifically selected because it ranked among the 10 worst-performing jobs and had a higher process count (512) and longer runtime (nearly 12 hours) than others in that group. In total, this job wrote 798 GiB to 122 different shared files using the MPI-IO interface but was only able



(a) For 50th percentile results, STDIO jobs access 2 KiB, POSIX jobs access 3 MiB, and MPI-IO jobs access 67 MiB. For 90th percentile results, STDIO jobs access 176 MiB, POSIX jobs access 6 GiB, and MPI-IO jobs access 21 GiB.



(b) For 50th percentile results, STDIO jobs access 2 files, POSIX jobs access 14 files, and MPI-IO jobs access 1 file. For 90th percentile results, STDIO jobs access 27 files, POSIX jobs access 47 files, and MPI-IO jobs access 20 files.



(c) For 50th percentile results, STDIO jobs attain 5.2 MiB/s, POSIX jobs attain 233.2 MiB/s, and MPI-IO jobs attain 570.9 MiB/s. For 90th percentile results, STDIO jobs attain 365.7 MiB/s, POSIX jobs attain 3.1 GiB/s, and MPI-IO jobs attain 2.6 GiB/s.

Figure 4: CDF plots of total bytes accessed, total files accessed, and observed performance using each interface across all jobs.

to sustain an average write bandwidth of 26.6 MiB/s. This performance pales in comparison with some of the highest-performing MPI-IO jobs, which were able to sustain over 240 GiB/s performance to the same Lustre file system used by this job.

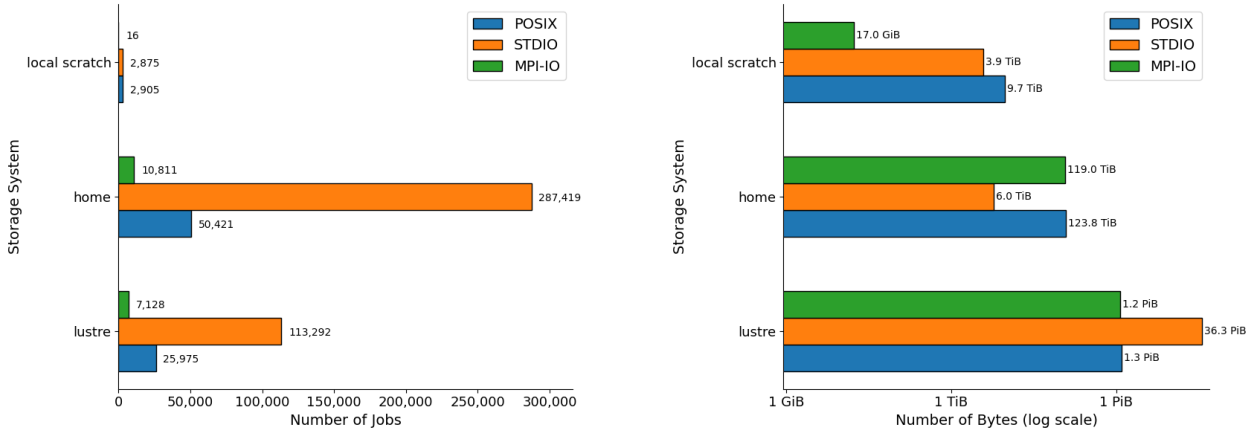
To gain initial insights into the I/O behavior of this job, we first consider the I/O activity heatmaps captured by Darshan. Heatmaps are a novel instrumentation technique in Darshan (introduced in version 3.4.0) aimed at providing higher-resolution temporal information on I/O workloads without resorting to more costly tracing. Darshan heatmaps capture per-rank I/O intensity over the duration of the application using histograms, illuminating potential hotspots and load imbalances that could negatively impact I/O performance. Currently, Darshan captures heatmaps for MPI-IO, POSIX, and STDIO interfaces, allowing for a better understanding of the relative intensity of I/O across these interfaces. Heatmaps for both MPI-IO and POSIX interfaces for this job are provided in Figure 6.

Application I/O phases are clearly visible for the MPI-IO heatmap in Figure 6a, with total I/O intensity evenly distributed across all ranks (as indicated by the marginal bars on the right y-axis). This is a typical pattern for MPI-IO collective I/O workloads. One obvious takeaway from this plot is that I/O phases appear to dominate the runtime of the application, which is not typical. Specifically, MPI-IO collective write operations account for over 70% of the application runtime, totaling over 8 hours of I/O activity. Improving the I/O performance of this workload could have a significant impact on this application by driving down the time spent doing I/O and, thus, the overall runtime.

To better understand the reasons for poor I/O performance for this job, we next consider the POSIX heatmap in Figure 6b. Differences between this heatmap and the previously analyzed MPI-IO heatmap are immediately obvious, as the intense MPI-IO activity evenly distributed across all ranks is transformed into an entirely different POSIX I/O pattern. More specifically, rank 0 is ultimately responsible for almost all POSIX I/O activity, performing write operations to the underlying Lustre storage system on behalf of all other ranks. This behavior understandably creates a serious performance bottleneck for this large-scale collective workload by failing to parallelize storage access.

While Darshan instrumentation does not explicitly capture the exact cause of this collective I/O bottleneck, there are enough context clues to isolate it to misconfigured Lustre stripe settings.[5] A well-known collective I/O optimization on Lustre file systems is to match the number of "aggregator" processes to the number of OST stripes [5], with this strategy adopted by the ROMIO MPI-IO implementation used by Cray MPICH on the ALCF Polaris system. Aggregator processes are the entities responsible for issuing I/O operations to the underlying storage system and redistributing results back to other processes. Thus, for shared files collectively accessed using MPI-IO on Lustre file systems, stripe settings directly affect the number of processes that can access storage in parallel. In the application from this case study, a stripe count of 1 likely was used (the default on Polaris's Lustre and Grand file systems), resulting in just a single process accessing storage (rank 0). This analysis is further supported by the fact that the majority of POSIX access

[5]Darshan has detailed Lustre instrumentation capabilities that explicitly capture stripe settings used on each file, but the Lustre module is not enabled on Polaris at time of this writing.



(a) Utilization of Polaris file systems according to total jobs.

(b) Utilization of Polaris file systems according to total bytes accessed.

Figure 5: Polaris file system utilization using different I/O interfaces.

sizes are 1 MiB, which is the default Lustre stripe size on Polaris, while MPI-IO level access sizes are generally much larger. In the Lustre optimization described above, MPI-IO level access sizes are broken down to align with the configured Lustre stripe size.

This case study highlights the importance of optimizing application and storage interactions to ensure efficient I/O workloads. Specifically in this case, using an increased Lustre stripe count could dramatically drive down the application runtime and increase overall system efficiency. HPC storage systems must support a variety of workloads, and default values are not necessarily intended to provide the best performance in all cases.

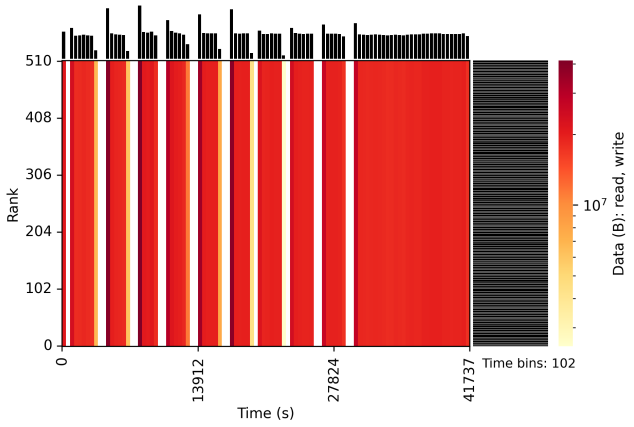
4.3.2 Case Study 2: extreme STDIO usage. Inspired by the staggering amount of STDIO I/O performed by outlier jobs discussed in Section 4.2, in this case study we take a closer look at the most extreme example. The job we chose to analyze is remarkable not only in that it totals over 529 TiB of total bytes read/written using STDIO APIs but also in that it is composed of just eight processes.

In Figure 7 we provide a high-level overview of the I/O cost for this application, focusing on the average cost of different I/O components (e.g., read, write, and metadata) using both STDIO and POSIX interfaces. The I/O costs of this application clearly are dominated almost entirely by STDIO read accesses, with read operations accounting for nearly 30% of the total 8-hour runtime. This is a considerable fraction of application runtime to spend on I/O, but it is not surprising given that each of the eight application processes was reading at least 66 TiB of data. STDIO write and metadata operations, as well as general POSIX I/O, are essentially negligible compared with these STDIO read operations.

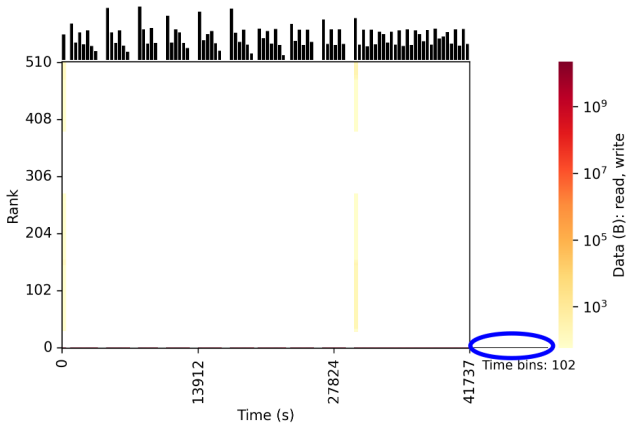
To gain a better understanding of the I/O access characteristics for this application, we provide details on the total number of STDIO operations issued in Figure 8. In total, the application issued over 28 million read operations, yielding an average read access size of just under 20 MiB. The most notable aspect of this data, however, is the sheer amount of write operations issued, especially given that read operations dominate total application I/O volume and runtime. Specifically, this application issues over 635 million write operations, amounting to approximately 21 GiB of data. The vast majority of these write operations are to a single 6 GiB file, yielding an average access size of just 10 bytes. Despite the inefficiency of such small, frequent writes to Lustre, the total write bandwidth to this output file reaches around 150 MiB/s. This performance is likely sustained due to aggressive buffering in memory by the C library, which helps mitigate the cost of excessive small I/O operations.

A deeper analysis of individual files accessed using STDIO revealed additional inefficiencies in this workload. First, we observed that the vast majority of STDIO read accesses were in the form of identical accesses (in terms of aggregate size and total operations) to per-rank files. Examining detailed per-file Darshan counters, we observed that each rank initially wrote approximately 1 GiB of data to its file at the start of execution. Throughout the remainder of the application's runtime, this same 1 GiB of data was repeatedly read, totaling over 60,000 times per rank. Offset statistics captured by Darshan indicate that no data was read beyond the offset of the initial write operation, suggesting that these files did not contain pre-existing data. This pattern of redundant I/O suggests an opportunity for optimization, where adjustments to data caching and reuse strategies could significantly reduce unnecessary file operations and improve overall performance. Aside from finding efficient data reuse strategies for these per-rank files, this workload is well suited for using either RAM or the local scratch file system given its limited size and frequent I/O operations, which can incur high latencies on Lustre file systems.

Further analysis of the 10-month Darshan log dataset revealed numerous jobs from this user exhibiting similar I/O behavior. To better understand their contribution to the unexpectedly high STDIO usage observed in Table 2, we compare this user's total STDIO activity with the aggregate usage across all jobs. As shown in Table 3, this user accounts for 97% of all bytes transferred via STDIO, despite representing only 0.09% of the total jobs. These findings highlight the disproportionate impact that individual outlier users can have on overall system I/O behavior. A previous analysis of systemwide Darshan data on the ALCF Intrepid system had similar findings,



(a) MPI-IO interface heatmaps illustrating numerous I/O phases. I/O intensity (measured in bytes) spans all ranks, as expected with MPI-IO collective operations. Bars on the upper x-axis indicate total sum of bytes across all ranks for a given time slice. Bars on the right y-axis indicate total sum across all time for a given rank, in this case indicating a balanced workload across all ranks.



(b) POSIX interface heatmaps illustrating essentially zero I/O activity on all ranks except rank 0. I/O activity for rank 0 is hard to visualize because it lies directly on the x-axis, but it is clear from the marginal bars on the right y-axis (circled in blue) that rank 0 accounts for essentially all I/O activity.

Figure 6: Heatmaps of application I/O activity across all ranks using MPI-IO and POSIX interfaces for the application from Case Study 1.

observing that a small number of users or projects contributed significantly to aggregate data access volumes [2].

4.4 Discussion

In this section we reflect on our results and discuss techniques for improving Darshan's usage on production systems and methods for optimizing the efficiency of Polaris applications.

Improving Darshan's coverage in production environments. As previous results have shown, Darshan's monthly coverage of Polaris

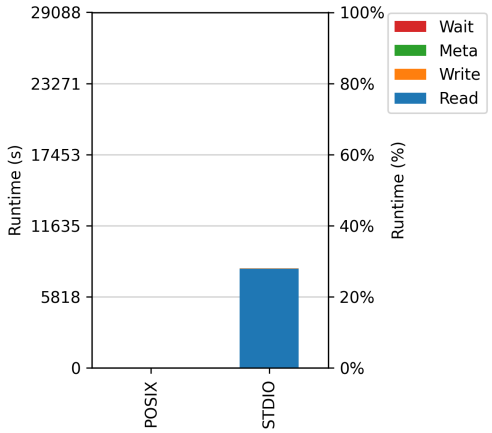


Figure 7: Average I/O cost across 8 processes of the application from Case Study 2 using STDIO and POSIX interfaces.

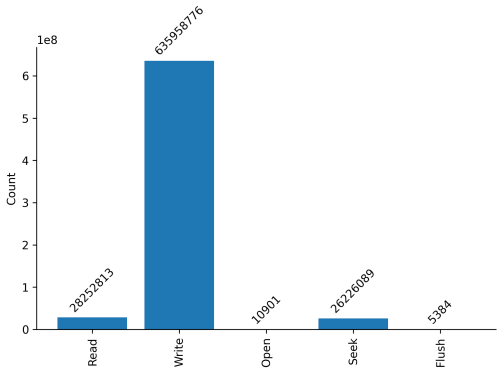


Figure 8: STDIO interface operation counts for Case Study 2, demonstrating the extreme number of operations issued by this application.

Job Set	Total Jobs	Total Bytes (PiB)	Total Files
All STDIO	404,604	36.66	16,147,980
User STDIO	355	35.57	266,981
User Percentage	**0.088%**	**97.03%**	**1.65%**

Table 3: Comparison of aggregate STDIO behavior across all jobs vs. jobs from the user in Case Study 2.

jobs averages around 20% of total node-hours, due to a number of factors. While some factors impacting coverage are outside of Darshan's control (most notably, applications explicitly choosing to disable Darshan), other factors could be mitigated by adjusting production deployments to take advantage of currently unused features.

For example, although Darshan was originally designed specifically for MPI applications, recent releases have expanded support for non-MPI use cases, including increasingly common Python-based applications such as those used in AI/ML. Despite this progress, these enhanced instrumentation capabilities are often not leveraged in production deployments like the one on Polaris, mostly because of changes in how Darshan instrumentation is enabled in this mode (using `LD_PRELOAD` at runtime rather than inserting instrumentation at build time). Closer collaboration with facilities to adopt these more flexible, runtime-based techniques would enable instrumentation of a broader range of emerging workloads, substantially increasing Darshan's coverage and utility.

As another example, Darshan has a long-standing feature that allows for instrumentation of applications that do not terminate cleanly (e.g., because they hit their wall-time limit and were aborted). This feature uses `mmap` to store Darshan instrumentation (rather than heap-allocated memory) so that the instrumentation can still be retrieved after processes terminate. In order to leverage this capability, the system must deploy a mechanism for reconstructing logs after jobs terminate (e.g., using job scheduler epilog scripts).

Working directly with facilities to help address these issues could lead to a significant boost in coverage, enabling more comprehensive understanding of system I/O behavior. Additionally, coordinating with high-impact applications or project teams could be another productive way to understand other blind spots in Darshan's instrumentation strategy and to help improve overall system coverage.

Early detection of inefficient, data-intensive workloads. A key takeaway from this work is that systemwide I/O behavior on production HPC systems can be dominated by a small subset of jobs, often corresponding to a particular user or project. Inefficiencies in these jobs' I/O workloads not only can degrade their own performance but also can adversely affect overall system behavior. Early detection mechanisms for identifying these jobs and inefficient aspects of their workloads can dramatically drive down application runtimes and improve overall system efficiency, benefiting all users.

The log collection workflow described in this work provides a natural mechanism for continuous monitoring of Polaris jobs to identify candidates for performance optimization. Additionally, our experience analyzing Darshan data and detecting common inefficiencies further simplifies the process of encoding detectors for poor I/O performance into an automated workflow. The output of such a workflow could be readily leveraged by facilities for early detection of jobs with inefficient I/O patterns and could guide targeted outreach efforts to improve both individual application behavior and overall system performance.

5 Conclusion

In this work we addressed an outstanding knowledge gap in the HPC I/O research community by developing an automated workflow for publishing Darshan log data collected from production applications running on the Polaris HPC system at the ALCF. We conducted a preliminary analysis of this dataset, highlighting general systemwide I/O trends and common I/O inefficiencies impacting Polaris applications. We also provided a discussion of best practices for collecting large-scale telemetry datasets and offered insights into the impact this data could have on application and systemwide I/O performance on HPC systems. Ultimately, we believe that expanding community access to these datasets will not only advance the understanding and optimization of scientific data access on HPC systems, but also enable collaborative analysis efforts that can yield actionable insights for facilities to incorporate into their own operational practices.

We plan to coordinate with facilities on effective strategies for continuously analyzing Darshan data to identify and address common I/O inefficiencies in data-intensive applications, essentially productizing our work here and integrating directly into facility operational workflows. We also aim to continue collaborating with facilities teams to capture similar datasets from other HPC systems that deploy Darshan in production (e.g., ALCF Aurora, NERSC Perlmutter and OLCF Frontier). These systems likely have distinct I/O access characteristics compared with the ALCF Polaris results presented here that are of high value to the research community. In particular, ALCF's Aurora system deploys DAOS, a novel object-based storage architecture, and understanding how scientific applications leverage this new storage technology will be critical. We additionally plan to work with facilities to augment anonymized Darshan data with additional project and application context, enabling more detailed analysis of I/O characteristics and trends within and across scientific domains. Automated capture, anonymization, and publication of other forms of system telemetry (e.g., network counters) would be a valuable contribution as well. We focus exclusively on Darshan I/O characterization in this work, but the core concepts are also applicable to other types of telemetry.

Acknowledgments

This research used resources of the Argonne Leadership Computing Facility, a U.S. Department of Energy (DOE) Office of Science user facility at Argonne National Laboratory and is based on research supported by the U.S. DOE Office of Science-Advanced Scientific Computing Research Program, under Contract No. DE-AC02-06CH11357. Data used in this work was generated from resources of the Argonne Leadership Computing Facility, which is a DOE Office of Science User Facility supported under Contract DE-AC02-06CH11357.

References

[1] Ryan Adamson, Paul Bryant, Dave Montoya, Jeff Neel, Erik Palmer, Ray Powell, Ryan Prout, and Peter Upton. 2024. Creating Continuous Integration Infrastructure for Software Development on US Department of Energy High-Performance Computing Systems. *Computing in Science & Engineering* 26, 1 (2024), 31–39.

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

[3] Philip Carns, Yushu Yao, Kevin Harms, Robert Latham, Robert Ross, and Katie Antypas. 2013. Production I/O characterization on the Cray XE6. In *Proceedings of the Cray User Group meeting*, Vol. 2013.

[4] Hariharan Devarajan, Loïc Pottier, Kaushik Velusamy, Huihuo Zheng, Izzet Yildirim, Olga Kogiou, Weikuan Yu, Anthony Kougkas, Xian-He Sun, Jae Seung Yeom, et al. 2024. DFTracer: An Analysis-Friendly Data Flow Tracer for AI-Driven Workflows. In *SC24: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–24.

[5] Phillip M Dickens and Jeremy Logan. 2010. A high performance implementation of MPI-IO for a Lustre file system environment. *Concurrency and Computation: Practice and Experience* 22, 11 (2010), 1433–1449.

[6] Théo Jolivel, François Tessier, Julien Monniot, and Guillaume Pallez. 2024. Mosaic: Detection and Categorization of I/O Patterns in HPC Applications. In *SC24-W: Workshops of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1311–1319.

[7] Glenn K Lockwood, Shane Snyder, Teng Wang, Suren Byna, Philip Carns, and Nicholas J Wright. 2018. A year in the life of a parallel file system. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 931–943.

[8] Jakob Luettgau, Shane Snyder, Tyler Reddy, Nikolaus Awtrey, Kevin Harms, Jean Luca Bez, Rui Wang, Rob Latham, and Philip Carns. 2023. 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*. 1380–1391.

[9] Huong Luu, Marianne Winslett, William Gropp, Robert Ross, Philip Carns, Kevin Harms, Mr Prabhat, Suren Byna, and Yushu Yao. 2015. A multiplatform study of I/O behavior on petascale supercomputers. In *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*. 33–44.

[10] Nafiseh Moti, André Brinkmann, Marc-André Vef, Philippe Deniel, Jesus Carretero, Philip Carns, Jean-Thomas Acquaviva, and Reza Salkhordeh. 2023. The I/O trace initiative: Building a collaborative I/O archive to advance HPC. In *Proceedings of the SC'23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis*. 1216–1222.

[11] Dmytro Povaliaiev, Radita Liem, Julian Kunkel, Jay Lofstead, and Philip Carns. 2024. High-Quality I/O Bandwidth Prediction with Minimal Data via Transfer Learning Workflow. In *2024 IEEE 36th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*. IEEE, 93–104.

[12] Shane Snyder, Philip Carns, Kevin Harms, Robert Ross, Glenn K Lockwood, and Nicholas J Wright. 2016. Modular HPC I/O characterization with Darshan. In *2016 5th workshop on Extreme-scale Programming Tools (ESPT)*. IEEE, 9–17.

[13] Shane Snyder and Argonne Leadership Computing Facility. 2025. *ALCF Polaris Darshan Log Collection*. doi:10.5281/zenodo.15052603

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