Enabling Agile Analysis of I/O Performance Data with PyDarshan

Jakob Luettgau
luettgauj@acm.org
Inria
Rennes, France

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

Tyler Reddy
treddy@lanl.gov
Los Alamos National Laboratory
Los Alamos, NM
USA

Nikolaus Awtrey
Los Alamos National Laboratory
Los Alamos, NM
USA

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

Jean Luca Bez
jlbez@lbl.gov
Lawrence Berkeley National Laboratory
Berkeley, CA, USA

Rui Wang
rwang@anl.gov
Argonne National Laboratory
Lemont, IL, USA

Rob Latham
robl@mcs.anl.gov
Argonne National Laboratory
Lemont, IL, USA

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

ABSTRACT

Modern scientific applications utilize numerous software and hardware layers to efficiently access data. This approach poses a challenge for I/O optimization because of the need to instrument and correlate information across those layers. The Darshan characterization tool seeks to address this challenge by providing efficient, transparent, and compact runtime instrumentation of many common I/O interfaces. It also includes command-line tools to generate actionable insights and summary reports. However, the extreme diversity of today's scientific applications means that not all applications are well served by one-size-fits-all analysis tools.

In this work we present PyDarshan, a Python-based library that enables agile analysis of I/O performance data. PyDarshan caters to both novice and advanced users by offering ready-to-use HTML reports as well as a rich collection of APIs to facilitate custom analyses. We present the design of PyDarshan and demonstrate its effectiveness in four diverse real-world analysis use cases.

KEYWORDS

High-Performance Computing, Storage, Performance Analysis, Input/Output

ACM Reference Format:
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 *Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis (SC-W 2023), November 12–17, 2023, Denver, CO, USA*. ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3624062.3624207

1 INTRODUCTION

Understanding how applications and workflows access data is an incredibly important aspect of computational workloads on HPC platforms. This importance is amplified by the increasing fidelity and rate of data produced by scientific instruments and simulations [9, 20, 40]. At the same time, new workloads are emerging that stress I/O systems differently from traditional applications [12, 22]. The variety of workloads present on modern platforms include simulations, experimental data analysis, inference and training of machine learning, and hybrid workloads incorporating elements of each. Many applications run into I/O bottlenecks that require understanding I/O behavior, especially as they scale. As a result, better tools for I/O analysis are vital. In the short term, such tools can directly improve application and workflow performance. In the long term, such tools can provide insight into compute, network, and storage utilization to inform the construction of efficient future systems.

Scientific applications use many different programming interfaces for I/O, from high-level interfaces for conveniently describing data to low-level interfaces that interact more directly with storage hardware. In many cases, data is translated through multiple layers on its way to a distributed storage system. This complexity makes analyzing and optimizing application and workflow I/O a daunting task requiring multilevel instrumentation. A common solution for the instrumentation of HPC applications is Darshan [18], which allows efficient instrumentation for profiling and tracing across many I/O layers such as POSIX, STDIO, MPI-IO, HDF5, and Lustre.

Up to now, the infrastructure for analyzing Darshan data has been limited to ad hoc scripts and C-coded tools. In this paper we present PyDarshan, a library of tools designed to enable agile analysis of I/O performance data captured using Darshan. Our contributions with PyDarshan are the following:

- Connect Darshan performance data to a rich Python ecosystem of data science, machine learning, and visualization libraries
- Promote interoperability of performance analysis tools by facilitating access to reusable analysis routines
- Enable more efficient access to Darshan data for the analysis of large collections of Darshan logs and longitudinal studies
- Present multiple use cases to illustrate how PyDarshan capabilities enable agile development of insightful I/O analysis tools

PyDarshan opens up the analysis of I/O behavior to a much broader audience, allowing a deeper understanding of the importance and characteristics of I/O in scientific computing today and

SC-W 2023, November 12–17, 2023, Denver, CO, USA

ACM ISBN 979-8-4007-0785-8/23/11…$15.00
https://doi.org/10.1145/3624062.3624207

in the future. PyDarshan is open source and available on GitHub[1] and PyPi.[2]

2 PYDARSHAN DESIGN

In this section we introduce the design of PyDarshan and discuss how it helps improve user experience, either through ready-to-use analysis tools or through interfaces enabling specific custom analysis needs. We begin the discussion with a brief overview of Darshan's existing architecture and then describe new analysis capabilities introduced by PyDarshan and how they facilitate I/O researchers and application teams to build interoperable tools or conduct custom analyses.

2.1 Darshan

Darshan [18] consists of two components, as illustrated in Figure 1: ❶ An *instrumentation runtime* collects data on application I/O activity and stores this data in Darshan log files, and ❷ a *collection of programmatic libraries and command-line (CLI) utilities* provide access to Darshan log data and perform a number of common analysis tasks.

The runtime component can instrument applications without needing to apply modifications to the application or library dependencies. Across various I/O layers, Darshan transparently collects statistics activity of HDF5, MPI-IO, STDIO, and POSIX interfaces by intercepting API calls (e.g., using `LD_PRELOAD`), as illustrated in Figure 1. For the Lustre parallel file system, Darshan collects additional context, such as the file mapping to object storage targets (OSTs). Darshan's runtime library is deliberately designed to minimize runtime and log storage overhead to avoid perturbing the application or the storage system [18]. Generally, Darshan data is captured independently on each process or rank. If MPI is used for parallel applications, data is gathered from all ranks at shutdown time and combined into a single Darshan log file. In case a process forks, independent Darshan log files will be generated for parent and children processes. Darshan defers writing instrumentation data until applications are shutting down by using MPI or Linux process life-cycle hooks.

Darshan log file data can be viewed using a collection of CLI utilities included with Darshan. These tools enable a range of common capabilities such as log parsing, data anonymization, and generating PDF summary reports. With growing applications, thousands of ranks, and growing bodies of Darshan logs collected by HPC sites, these analysis tools increasingly run into performance and memory constraints. The legacy approach for generating PDF report relies on a complex analysis pipeline. Logs are translated into an intermediate text format, parsed and analyzed in Perl, and then transformed into user-facing document with TeX. This served as an effective initial exploration of how to present I/O analysis data to users, but it is not as efficient or flexible as new approaches enabled by a now expansive, feature-rich Python software ecosystem.

2.2 PyDarshan

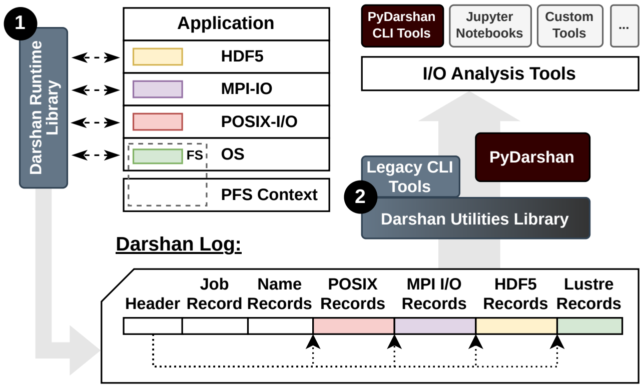


Figure 1: Existing architecture of Darshan, its log format, and post-processing utilities. This includes PyDarshan as well as extensions to facilitate custom tools or analysis in Jupyter Notebooks. While most Darshan modules feature only integer and floating-point counters, other modules have been added to capture traces or additional system information.

PyDarshan extends Darshan's existing analysis capabilities with a convenient Python interface and corresponding CLI utilities. The primary goal of PyDarshan is to enable advanced and customized analysis that can be connected seamlessly to the rich ecosystem of data science and machine learning libraries that support Python. PyDarshan caters to many concepts from popular data science libraries such as Numpy[23], Pandas[31], or XArray[24], which many users are familiar with already and for which data science and machine learning libraries provide tooling to consume for analysis and model training. In the spirit of testing the new tools ourselves, we migrated our report generation to using the Python ecosystem that informed PyDarshan's architecture. Instead of using PDF reports, we switched to HTML-based reports, which are easier to customize and are just as widely supported on most systems. While the new reports are part of PyDarshan's architecture, this section will focus on the overall architecture, and we will revisit the updated report generation in Section 3.1.

Figure 2 illustrates the architecture of PyDarshan. PyDarshan internally uses the same Darshan C library to parse logs that is also used by the original command-line utilities. A side effect of PyDarshan's development is also the streamlining of the C library for use by third-party tools. As a result, PyDarshan's contributions span three layers to interact with Darshan logs. We extend Darshan's C utility library and introduce two new APIs that progressively offer more control to trade off efficiency and implementation effort:

- ❸ The *High-Level PyDarshan Object Interface* provides dedicated objects for loaded data such as defined by the DarshanReport and RecordCollections classes that offer introspection, automatic statistics, report generation, and conversion to data formats that allow easy reshaping for use with custom tools or data science and machine learning libraries such as scikit-learn, PyTorch, or TensorFlow.
- ❹ The *Low-Level PyDarshan Log Backend Interface* abstracts away Darshan-specific implementation details by providing thin wrappers around common functionality such as accessing metadata, name records, and module data. The low-level PyDarshan

[1] https://github.com/darshan-hpc/darshan
[2] https://pypi.org/project/darshan/

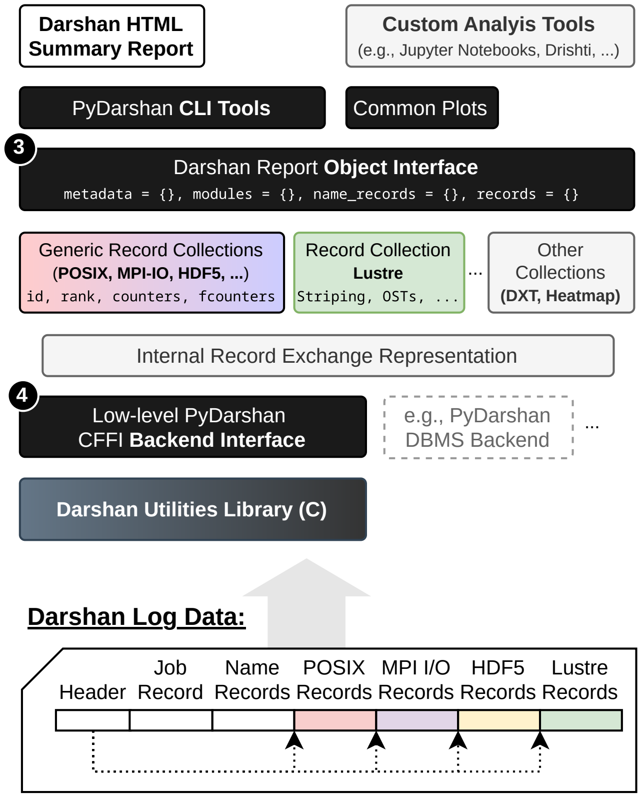

Darshan HTML Summary Report
Custom Analysis Tools
(e.g., Jupyter Notebooks, Drishti, ...)
PyDarshan CLI Tools
Common Plots
3
Darshan Report Object Interface
metadata = {}, modules = {}, name_records = {}, records = {}
Generic Record Collections
(POSIX, MPI-IO, HDF5, ...)
id, rank, counters, fcounters
Record Collection
Lustre
Striping, OSTs, ...
...
Other Collections
(DXT, Heatmap)
Internal Record Exchange Representation
4
Low-level PyDarshan CFFI Backend Interface
e.g., PyDarshan DBMS Backend
...
Darshan Utilities Library (C)
Darshan Log Data:
Header
Job Record
Name Records
POSIX Records
MPI I/O Records
HDF5 Records
Lustre Records

Figure 2: Core components of PyDarshan, including the High-level Interface, the Low-Level Python Interface, and the updated C utility library to access the binary data.

log backend interface takes responsibility for translating Darshan's custom binary data format into Pythonic representations such as dictionaries or Pandas dataframes.

The *low-level Darshan C utility library* is a low-level interface whose primary responsibility is to extract raw data from Darshan logs. The Darshan runtime library emits logs in a compressed, write-optimized, binary format that emphasizes the minimization of runtime instrumentation overhead above all other considerations. A native C library implementation is the most straightforward way to directly extract this data back into memory for subsequent analysis. It also provides rudimentary aggregation capabilities that can be performed efficiently as an inline component of data extraction. The low-level C library interface is more complex and fragile than the Pythonic APIs presented by PyDarshan, however, and we therefore discourage its direct use in analysis utilities.

2.2.1 High-Level Python Object Interface. The High-Level Object API aims to provide convenient and efficient access to Darshan performance data and metadata while catering to context-aware aggregation functions that are sensitive to the intricacies of performance data in distributed storage systems. The High-Level Object API is further designed to allow interactive exploration of Darshan log data, for example, in Jupyter Notebook environments or in custom tools.

To avoid reimplementing common filtering and reduction functionality that already provide efficient or accelerated implementations, PyDarshan wraps log data into two primary object structures that can be easily inspected through both standard Python facilities and a special `info()` method:

- *Darshan Report Objects* are context-preserving containers for multilevel performance data on which we can define a wide range of common performance analyses
- *Darshan Record Collections* are iterable homogeneous containers for data of the same type on which we can define transformations and analysis that apply for particular record types

In particular, for record collections, we introduce `to_df()` and `to_dict()` to convert Darshan log data into types that can be fed into Python's rich existing ecosystem of data science libraries. A basic example illustrating PyDarshan's usage is shown in Listing 1.

```
import darshan

# open a Darshan log file and read all data
with darshan.DarshanReport(filename, read_all=True) as report:
  # print the metadata dict for this log
  print("metadata: ", report.metadata)
  # print job runtime and nprocs
  print("run_time: ", report.metadata['job']['run_time'])
  print("nprocs: ", report.metadata['job']['nprocs'])

  # print modules contained in the report
  print("modules: ", list(report.modules.keys()))

  # export POSIX module records to DataFrame and print
  posix_df = report.records['POSIX'].to_df()
  display(posix_df)
```

Listing 1: Example of PyDarshan's High-Level Object API.

We initially investigated loading most data into, for example, Pandas DataFrames directly instead of wrapping them into custom object types. Unfortunately, various instrumentation modules use nested data structures that would invalidate standard aggregations or increase PyDarshan's memory footprint significantly when using a flat tabular representation. Even though most Darshan logs primarily contain Darshan's generic log records that are already tabular in nature, an increasing number of modules require deeper nested structures to capture sufficient context. Examples are Darshan's eXtended Tracing [39] or the Lustre module that records how files are mapped to the OSTs of the Lustre parallel file system. Nested structures in DataFrames are often inconvenient to access, and flattened representations would introduce significant memory overhead. As we will explain later, directly converting to Pandas DataFrames also would have resulted in a significant performance penalty, contradicting one of the purposes of PyDarshan: allowing more convenient as well as fast and memory-efficient access to Darshan log data.

Since Darshan log records are usually compressed, the uncompressed data in Python could exceed the main memory available on the host. For these occasions, we offer the `read_all=False` switch when instantiating a report from a Darshan log. This suppresses loading the log records for all modules as well as loading any name records. In this mode, only metadata is loaded, enabling the inspection of logs with many billions of names or log records. The user then has the choice to selectively load only specific modules or to use an iterator that returns only one log record at a time.

We started the curation of specific aggregation functions and common plots guarded against performing illegal reductions. Many of PyDarshan's aggregation helpers are designed to be useful beyond single log analysis and instead target the aggregation and visualization of record collections mixed from multiple Darshan logs, as we will demonstrate in Section 3.

2.2.2 Low-Level PyDarshan Log Backend Interface. The Low-Level PyDarshan Log Backend Interface primarily serves as a helper for the High-Level Object Interface and is responsible for loading metadata and records from a Darshan log file. It also serves users who want to avoid the overhead of using the High-Level API. Its API closely aligns with the Darshan C utility library but instead returns Darshan log data in Python-friendly Numpy/Pandas/Dictionary representations while granting the same fine-grained control for all interactions with the log. A basic example illustrating PyDarshan's Log Backend API is shown in Listing 2.

For the Log Backend Interface, PyDarshan leverages Python's C Foreign Function Interface (CFFI), which allows calling into existing C libraries from Python. This often requires considerable boilerplate to transform between C and Python data structures, which the Log Backend Interface effectively hides from users. CFFI in Python can be used in different modes. At this time, we are using CFFI'S ABI in-line mode, which has the benefit of not requiring to build dependencies such as compilers and linkers. Instead, merely a header file and the shared library `libdarshan-util.so` are needed, which are now part of all darshan-utils installations by default.

PyDarshan currently ships only with the CFFI-based Log Backend, but we found the decoupling of the high-level API as a standard interface to base visualizations and common analysis useful. We focused on the PyDarshan Log Backend because at the moment Darshan data is typically read from Darshan log files directly. For analysis, that has to aggregate across an entire site, a project, or a particular user; however, this approach can have significant drawbacks because every log file has to be opened and inspected to see whether it matches a search criterion. As database-like metric stores are becoming the norm for many deployments, we also anticipate alternative backends.

```
import darshan.backend.cffi_backend as darshanll

log = darshanll.log_open("example.darshan")

# Access various job information
darshanll.log_get_job(log)
# Example Return:
# {'jobid': 4478544, 'uid': 69615,
  # 'start_time': 1490000867, 'end_time': 1490000983,
  # 'metadata': {'lib_ver': '3.1.3', 'h': 'romio_no_indep_rw=true;cb_nodes=4'}}

# Access available modules and modules
darshanll.log_get_modules(log)
# Example Return:
# {'POSIX': {'len': 186, 'ver': 3, 'idx': 1},
  #  'MPI-IO': {'len': 154, 'ver': 2, 'idx': 2},
  #  'LUSTRE': {'len': 87, 'ver': 1, 'idx': 6},
  #  'STDIO': {'len': 3234, 'ver': 1, 'idx': 7}}

# Access different record types as numpy arrays, with integer and float counters separated
# Example Return: {'counters': array([...], dtype=uint64), 'fcounters': array([...])}
posix_record = darshanll.log_get_record(log, "POSIX")
mpiio_record = darshanll.log_get_record(log, "MPI-IO")
stdio_record = darshanll.log_get_record(log, "STDIO")
# ...

darshanll.log_close(log)

```

Listing 2: Example using PyDarshan Log Backend API.

2.2.3 C Utility Library Extensions. To achieve efficient access to Darshan log data and avoid duplication of parsing logic in two code bases, PyDarshan extends and refactors some of Darshan's original C utility library used by Darshan's original command-line tools. Most of the API extensions grant access to otherwise opaque data structures used only internally by Darshan utilities (such as the hash tables used to map relations) so they can be easily consumed and transformed for access from Python and other external tools. In addition, facilities for fast data aggregation have been added for generic log records. These low-level C interface extensions were merged back into the original Darshan C utility suite.

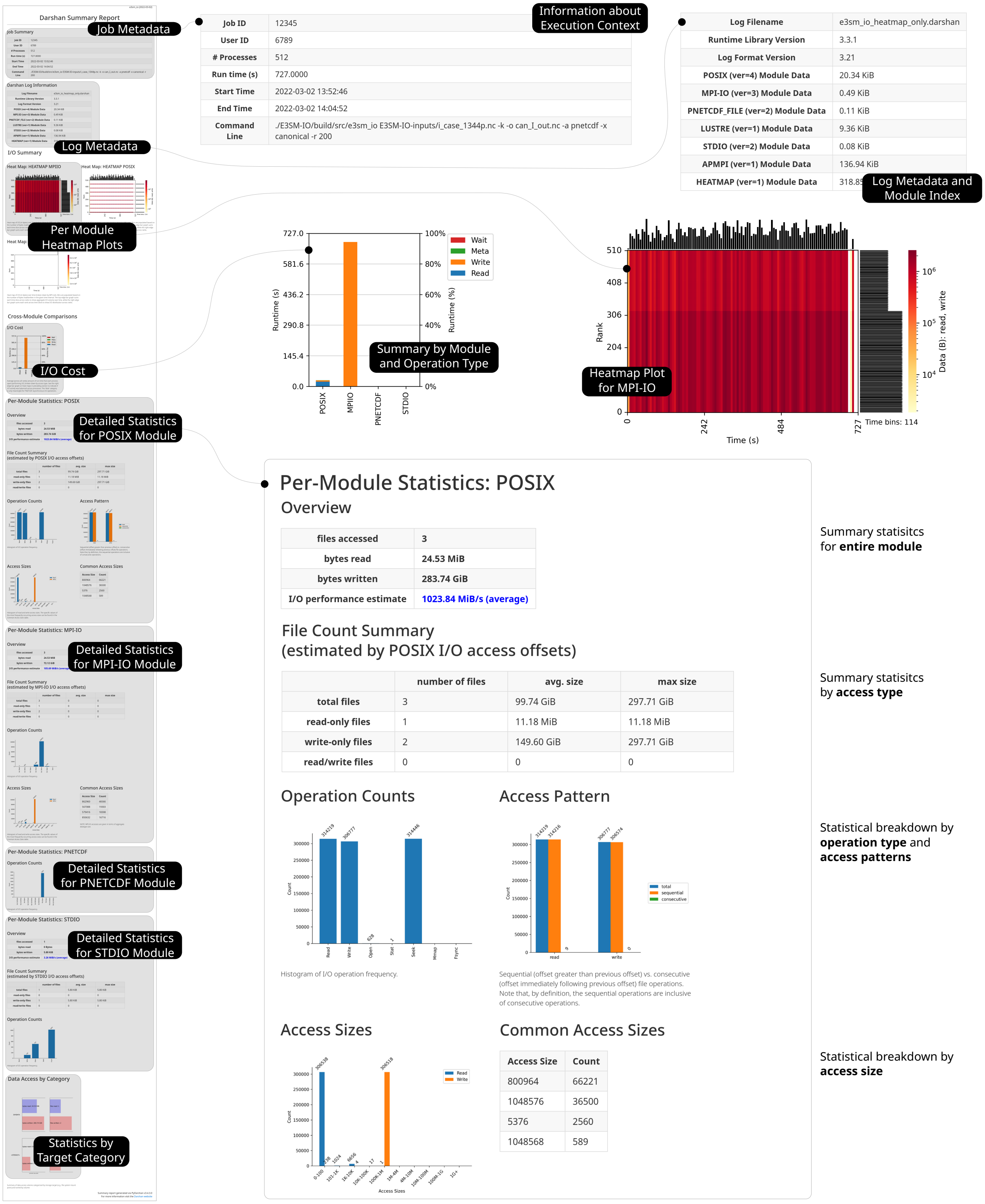
We have identified additional optimizations that could be implemented in the C library to better cater to the needs of PyDarshan in future releases. In particular, we are looking to facilitate batched and lazy loading capabilities that would benefit many data science frameworks offering out-of-core computations as well as shuffling data loaders often used in the context of machine learning. We will continue the discussion of these and other planned changes and future work in Section 5.

2.2.4 Python Packaging. We initially planned to keep PyDarshan a Python-only package, expecting most users to use PyDarshan in the same context in which the logs were generated, namely, the HPC cluster. For this reason, the PyDarshan package in the Python Package Index (PyPi) initially did not include a binary distribution, to avoid problems across different target architectures. But because many users reported that they would install PyDarshan on their personal computers, we have since started providing binary wheels that ship together with a precompiled copy of `libdarshan-util.so`, thus allowing use of PyDarshan on most platforms, either through the provided binary wheels for Linux and MacOS or using Windows Subsystem for Linux. This capability is particular helpful for fostering engagement with new researchers and students who may have access to Darshan logs but not the original HPC platform on which they were generated.

3 ANALYSIS CASE STUDIES

In this section we discuss four use cases, developed by teams at different organizations, that demonstrate the kinds of new analysis and tools that are enabled by PyDarshan:

- Use Case 1: Enhancing single job summaries with HTML reports and modular templates for more interactivity using a large-scale run of the E3SM [20] climate code in Section 3.1
- Use Case 2: Enabling custom analysis tools building on top of Darshan using the examples of DXT Explorer [11] and Drishti [10] in Section 3.2


Darshan Summary Report
Job Metadata
Log Metadata
Per Module Heatmap Plots
I/O Cost
Detailed Statistics for POSIX Module
Detailed Statistics for MPI-IO Module
Detailed Statistics for PNETCDF Module
Detailed Statistics for STDIO Module
Statistics by Target Category
Information about Execution Context
Log Metadata and Module Index
Summary by Module and Operation Type
Heatmap Plot for MPI-IO
727.0
581.6
436.2
290.8
145.4
0.0
Runtime (s)
100%
80%
60%
40%
20%
0%
Runtime (%)
Wait
Meta
Write
Read
POSIX
MPIIO
PNETCDF
STDIO
510
408
306
204
0
Rank
0
242
484
727
Time (s)
Time bins: 114
10^6
10^5
10^4
Data (B): read, write
Summary statisitcs for **entire module**
Summary statisitcs by **access type**
Statistical breakdown by **operation type** and **access patterns**
Statistical breakdown by **access size**

Job ID	12345
User ID	6789
# Processes	512
Run time (s)	727.0000
Start Time	2022-03-02 13:52:46
End Time	2022-03-02 14:04:52
Command Line	./E3SM-IO/build/src/e3sm_io E3SM-IO-inputs/i_case_1344p.nc -k -o can_I_out.nc -a pnetcdf -x canonical -r 200

Log Filename	e3sm_io_heatmap_only.darshan
Runtime Library Version	3.3.1
Log Format Version	3.21
POSIX (ver=4) Module Data	20.34 KiB
MPI-IO (ver=3) Module Data	0.49 KiB
PNETCDF_FILE (ver=2) Module Data	0.11 KiB
LUSTRE (ver=1) Module Data	9.36 KiB
STDIO (ver=2) Module Data	0.08 KiB
APMPI (ver=1) Module Data	136.94 KiB
HEATMAP (ver=1) Module Data	318.85

Per-Module Statistics: POSIX

Overview

files accessed	3
bytes read	24.53 MiB
bytes written	283.74 GiB
I/O performance estimate	1023.84 MiB/s (average)

File Count Summary
(estimated by POSIX I/O access offsets)

	number of files	avg. size	max size
total files	3	99.74 GiB	297.71 GiB
read-only files	1	11.18 MiB	11.18 MiB
write-only files	2	149.60 GiB	297.71 GiB
read/write files	0	0	0

Operation Counts

Histogram of I/O operation frequency.

Access Pattern

Sequential (offset greater than previous offset) vs. consecutive (offset immediately following previous offset) file operations. Note that, by definition, the sequential operations are inclusive of consecutive operations.

Access Sizes

Common Access Sizes

Access Size	Count
800964	66221
1048576	36500
5376	2560
1048568	589

Figure 3: Overview of the PyDarshan job summary report. A zoomed-out view of the full HTML report is shown on the left while key sections are highlighted on the right.

- Use Case 3: Customizing I/O analysis of workflows using the example of ATLAS AthenaMP, a high-energy physics simulation in Section 3.3
- Use Case 4: Enabling the analysis of large bodies of Darshan logs with hundreds of thousands of jobs on the Cori and Theta supercomputers in Section 3.4

3.1 Use Case 1: Enhancing Single Job Summaries with New Views

In this first use case, we discuss how PyDarshan enhances the capabilities of the existing Darshan job summary tool and lowers the burden for users to contribute. Specifically, building off the success of the Python community, we can leverage diverse packages to simplify the process of maintaining and extending Darshan summary report infrastructure (e.g., for interfacing with Darshan utility libraries, plotting log data, generating report HTML code from templates), leading to a much more developer-friendly experience.

The Darshan job summary tool is an important starting point for many users because the detailed data captured by Darshan is often cumbersome for users to aggregate and analyze directly. For example, users are often concerned with high-level insights into I/O behavior such as quantifying attained performance or detecting potential bottlenecks, yet this information is not always straightforward to deduce from the raw instrumentation data captured by Darshan. Analysis tools that can automatically extract log data and summarize key insights into I/O behavior are critical to ensuring that users can take advantage of the detailed I/O characterization data Darshan provides. These tools can lead users to more readily recognize and address I/O inefficiencies in their software, ideally leading to more efficient usage of HPC systems and increased scientific productivity.

Traditionally, Darshan has provided a Perl-based job summary script to help summarize key I/O characteristics of a given Darshan log file. This tool extracts relevant I/O data from an input log file, aggregates this data across all instrumented files/processes, and produces a PDF summarizing I/O behavior with graphs, tables, annotations, and so on. Although this tool has proven helpful for Darshan users looking to quickly gain a high-level overview of the I/O behavior of their jobs, it has several shortcomings that limit its efficiency and extensibility:

- Inefficient log data extraction relying on the raw text output of the `darshan-parser` utility rather than the binary data format provided by the Darshan utility library
- Lack of an extensible summary report template, instead using a fixed LaTeX-based template, hindering the ability to extend or otherwise reorganize summary reports
- Lack of an in-house plotting library, instead relying on manually copying relevant log data to files that can then be fed to `gnuplot` scripts

To address these problems, we have completely reimplemented the Darshan job summary tool to leverage PyDarshan and other auxiliary Python packages. This tool leans heavily on the common PyDarshan infrastructure introduced in Section 2.2. To start, a Darshan report object is instantiated that reads in all log data and metadata required by the summary tool using bindings to the Darshan C utility library. Next, the report content (figures, tables, annotations) is generated from relevant log data, typically using PyDarshan routines for producing common plots (using Matplotlib and Seaborn) and tables (using Pandas). This report content is then converted to HTML (e.g., using Base64 encoding for Matplotlib figures, using Pandas `to_html()` method for tables) and registered within different sections in the overall report structure. A Mako[3] HTML template is used to programmatically integrate all report content generated by the job summary tool into a single output HTML report, with CSS grids used to help enforce organization and styling of different sections/figures.

An overview of an example PyDarshan job summary report is presented in Figure 3. The first section of the report contains important metadata related to the job itself (date, runtime, command-line args, etc.) and the corresponding Darshan log file (log file name, active modules, etc.). Additional report sections provide information on overall job I/O activity and I/O cost. Job I/O activity is summarized using heatmap figures that indicate I/O intensity in terms of total bytes read/written across ranks over time, with these figures included for multiple interfaces. These heatmap figures can be generated either from Darshan's new HEATMAP module or from DXT trace data. I/O cost is presented as the average per-process cost of different I/O components (read, write, metadata) relative to application runtime. Subsequent sections contain detailed per-module statistics allowing deeper insights into usage of different interfaces and corresponding performance characteristics. The final section provides details on the application's overall usage of different storage targets (file system mounts, standard streams, etc.).

3.2 Use Case 2: Enabling Custom Analysis Views and Tools for Applications

In the second use case we discuss how PyDarshan facilitates the development of custom tools such as DXT Explorer [11] and Drishti I/O [8]. DXT Explorer is a powerful tool to navigate data provided by Darshan's eXtended Tracing (DXT) module. Drishti is a novel interactive, user-oriented visualization and analysis framework designed to face the existing challenges in understanding I/O metrics.

Identifying the root causes of I/O performance inefficiencies in scientific applications requires a combination of detailed metrics and an understanding of the complex HPC I/O stack. Despite the numerous tools that collect I/O metrics on production supercomputer systems, applications often require an I/O specialist to detect the

[3] https://www.makotemplates.org/

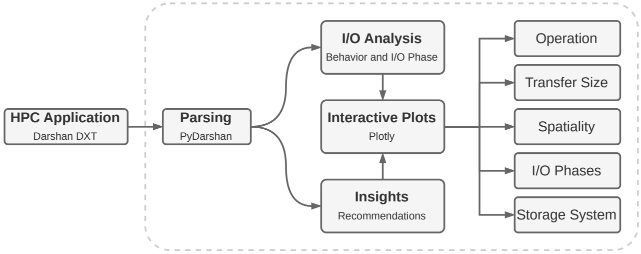


Figure 4: Drishti generates meaningful interactive visualizations and a set of recommendations based on the detected I/O bottlenecks using PyDarshan to handle Darshan data.

root causes of performance bottlenecks when accessing data and determine what to do to solve them. For end users, a gap exists between the currently available metrics, the issues they represent, and the application of optimizations that would mitigate slowdowns.

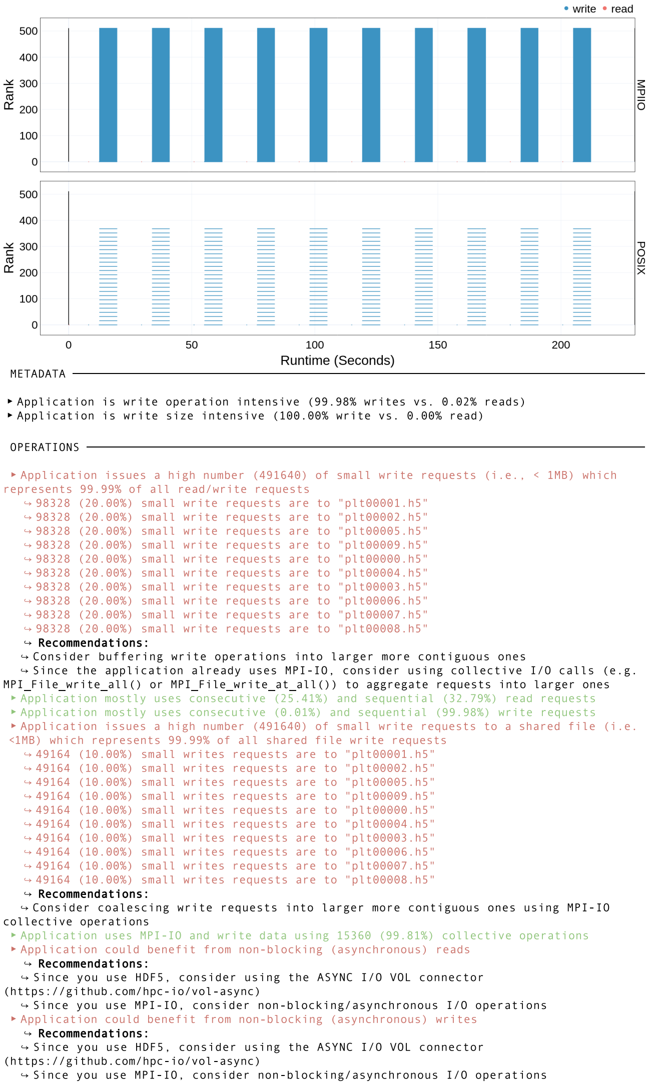
Drishti[4] is a novel interactive, user-oriented visualization and analysis framework designed to face the existing challenges in understanding I/O metrics [8]. It combines the features from DXT Explorer and Drishti I/O. From extracting I/O behavior and illustrating it for users to explore interactively, detecting I/O bottlenecks automatically, and presenting a set of recommendations to avoid them, Drishti seeks to streamline the process of identifying and fixing I/O bottlenecks. The framework relies on automatically detecting common root causes of I/O performance inefficiencies by mapping raw metrics into problems and providing natural language feedback to users based on heuristics. Figure 4 illustrates Drishti's core components.

To avoid multiple conversions and overhead, instead of using the Darshan command-line parser and then transforming that to a CSV so data could be ingested by Drishti, we instead rely on PyDarshan for direct access to Darshan's counters and DXT trace data in the form of Pandas DataFrames. PyDarshan returns a dictionary of DataFrames containing all trace operations issued by each rank. This data structure is not optimal for Drishti because it requires an overall view of the application behavior. Hence, we take an additional step to iterate through the dictionary of DataFrames and merge them into a single DataFrame for analysis and interactive visualization.

To illustrate Drishti's core features, we analyze the AMReX [40] application that relies on highly parallel adaptive mesh refinement (AMR) algorithms to solve partial differential equations on block-structured meshes. We ran AMReX with 512 ranks over 32 nodes on the Cori supercomputer at NERSC, with a 1024 domain size, a maximum allowable size of each subdomain used for parallel decomposal as 8, 1 level, 6 components, 2 particles per cell, 10 plot files, and a sleep time of 10 seconds between writes.

Figure 5 depicts the baseline execution of AMReX with the triggered issues and Drishti's recommendations. Because AMReX relies on the HDF5 data format and it interleaves computation with I/O phases, we have integrated the asynchronous I/O VOL Connector [34] in an effort to hide some of the time spent in I/O through nonblocking operations while the application makes progress in its computation. Drishti reports also highlight that most write requests that arrive at the parallel file system are < 1 MB for all 10 plot files. Because the application issues larger requests that are broken down into smaller ones directed at each storage server based on the striping policy, we have increased the stripe size to 16 MB. By applying such optimizations we achieved a total speedup of 2.1× (from 211 to 100 seconds).

Drishti can also be used to provide insights into the overall user behavior in a supercomputer, which we will revisit in Section 3.4 where we investigate how PyDarshan enables the analysis of large bodies of Darshan logs.



```
 METADATA ─────────────────────────────────────────────────────────────────

 ▸ Application is write operation intensive (99.98% writes vs. 0.02% reads)
 ▸ Application is write size intensive (100.00% write vs. 0.00% read)

 OPERATIONS ───────────────────────────────────────────────────────────────

 ▸ Application issues a high number (491640) of small write requests (i.e., < 1MB) which
represents 99.99% of all read/write requests
   ↪ 98328 (20.00%) small write requests are to "plt00001.h5"
   ↪ 98328 (20.00%) small write requests are to "plt00002.h5"
   ↪ 98328 (20.00%) small write requests are to "plt00005.h5"
   ↪ 98328 (20.00%) small write requests are to "plt00009.h5"
   ↪ 98328 (20.00%) small write requests are to "plt00000.h5"
   ↪ 98328 (20.00%) small write requests are to "plt00004.h5"
   ↪ 98328 (20.00%) small write requests are to "plt00003.h5"
   ↪ 98328 (20.00%) small write requests are to "plt00006.h5"
   ↪ 98328 (20.00%) small write requests are to "plt00007.h5"
   ↪ 98328 (20.00%) small write requests are to "plt00008.h5"
   ↪ Recommendations:
  ↪ Consider buffering write operations into larger more contiguous ones
  ↪ Since the application already uses MPI-IO, consider using collective I/O calls (e.g.
MPI_File_write_all() or MPI_File_write_at_all()) to aggregate requests into larger ones
 ▸ Application mostly uses consecutive (25.41%) and sequential (32.79%) read requests
 ▸ Application mostly uses consecutive (0.01%) and sequential (99.98%) write requests
 ▸ Application issues a high number (491640) of small write requests to a shared file (i.e.
 <1MB) which represents 99.99% of all shared file write requests
   ↪ 49164 (10.00%) small writes requests are to "plt00001.h5"
   ↪ 49164 (10.00%) small writes requests are to "plt00002.h5"
   ↪ 49164 (10.00%) small writes requests are to "plt00005.h5"
   ↪ 49164 (10.00%) small writes requests are to "plt00009.h5"
   ↪ 49164 (10.00%) small writes requests are to "plt00000.h5"
   ↪ 49164 (10.00%) small writes requests are to "plt00004.h5"
   ↪ 49164 (10.00%) small writes requests are to "plt00003.h5"
   ↪ 49164 (10.00%) small writes requests are to "plt00006.h5"
   ↪ 49164 (10.00%) small writes requests are to "plt00007.h5"
   ↪ 49164 (10.00%) small writes requests are to "plt00008.h5"
   ↪ Recommendations:
  ↪ Consider coalescing write requests into larger more contiguous ones using MPI-IO
collective operations
 ▸ Application uses MPI-IO and write data using 15360 (99.81%) collective operations
 ▸ Application could benefit from non-blocking (asynchronous) reads
   ↪ Recommendations:
  ↪ Since you use HDF5, consider using the ASYNC I/O VOL connector
(https://github.com/hpc-io/vol-async)
  ↪ Since you use MPI-IO, consider non-blocking/asynchronous I/O operations
 ▸ Application could benefit from non-blocking (asynchronous) writes
   ↪ Recommendations:
  ↪ Since you use HDF5, consider using the ASYNC I/O VOL connector
(https://github.com/hpc-io/vol-async)
  ↪ Since you use MPI-IO, consider non-blocking/asynchronous I/O operations
```

Figure 5: Interactive visualization of Darshan's DXT MPI-IO and DXT POSIX trace information and Drishti's recommendations report for AMReX.

3.3 Use Case 3: Enabling Custom Analysis for Scientific Workflows

In this third use case we present the custom tooling on top of PyDarshan that was developed by an application team to track the I/O performance of a production high-energy physics workflow. The example shows how existing PyDarshan analysis capabilities, such as the POSIX access size plot and heatmap, can be repurposed to provide insights on entire workflows rather than individual jobs. It also demonstrates that many teams require insights and custom plots that out-of-the-box tools cannot anticipate.

The ATLAS experiment [9] at the Large Hadron Collider (LHC) at CERN studied here uses Athena [1] as its main simulation, reconstruction, and analysis software framework and uses ROOT [16] for its I/O and storage.

[4] https://github.com/hpc-io/drishti

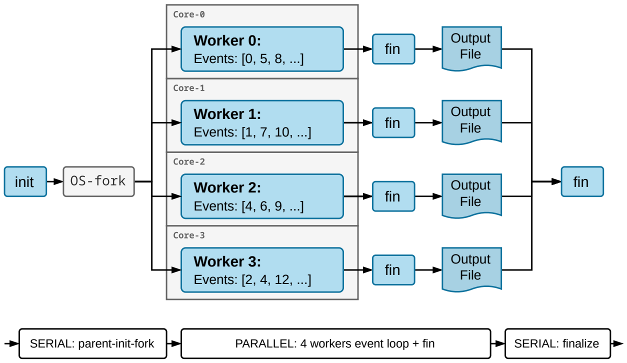


Figure 6: Schematic of the Atlas AthenaMP [17] workflow with events distributed across 4 workers. The serial and parallel phases of the workflow are indicated at the bottom.

Athena was initially written to run serially. It was then extended to support multiprocess parallelism for Run 2,[5] so-called AthenaMP, where independent parallel workers are forked from the main process with a shared-memory allocation (see Figure 6). In this version of the workflow, each worker produces its own output file. This proves inefficient, however, because these individual worker files have to eventually undergo a serial merging process, requiring them each to be reread from file. In order to increase the efficiency of this process, a SharedWriter process has been designed to be executed alongside the other workers that retrieves the output data objects from the workers' memory and merges them on the fly. Parallel compression was also implemented to further improve efficiency after observing large overheads when scaling up to higher process counts. To assist in these types of performance-tuning efforts going forward, Darshan is now being used to monitor various I/O performance factors in Athena workflows.

To track load imbalances across the different workers and writers in the workflow stages that use AthenaMP, heatmap records are extracted from each Darshan log generated from the main/forked processes. These heatmaps are aligned and combined into a single heatmap DataFrame with the same format of what is used by the PyDarshan heatmap plotting function, although with a slight modification. Instead of MPI ranks, the y-axis references different logical units such as the workers and the shared writer of the workflow, as illustrated in Figure 7, allowing for a comprehensive view of the I/O activity of the entire workflow. The performance impacts of previously discussed tuning decisions can be clearly observed in Figure 7 (left), where the SharedWriter process spends the majority of its time writing compared with Figure 7 (right) where the parallel compression helps alleviate this bottleneck by reducing the frequency of writing in the SharedWriter process. In this way, these heatmap plots helped the team visualize and quickly spot load imbalances between workers and SharedWriter in AthenaMP.

Understanding the breakdown of I/O operations by type (such as read, write, open, stat, seek, mmap, and fsync) for each of the workflow components is also useful for gaining insights into various AthenaMP configurations. To do so, PyDarshan was used to read log records associated with workflow input and output files into a DataFrame for both main and forked processes. Then, aggregations are performed on each type of I/O operations before the data is combined and fed into the custom stacked bar plot shown in Figure 8. Because the total number of the operations could differ by a several orders of magnitude, the fraction of each kind of operation is also displayed in the bottom panel. This experiment processes 3,600 events per process and aggregates the total operations for each of the workflow components (main, shared writer, workers) by operation type. Figure 8 (left) demonstrates that all of the writes are performed by the SharedWriter process while none are performed by the workers. As expected, the workers all share the same general I/O behavior characteristics. An equal number of writes and seeks have been found in the SharedWriter process. There are about twice as many seeks as reads in the workers, a result that requires further understanding. Figure 8 (right) shows a small amount of extra reads in the SharedWriter process when parallel compression is enabled.

3.4 Use Case 4: Analyzing Large Quantities of Logs for System-Wide I/O Behavior

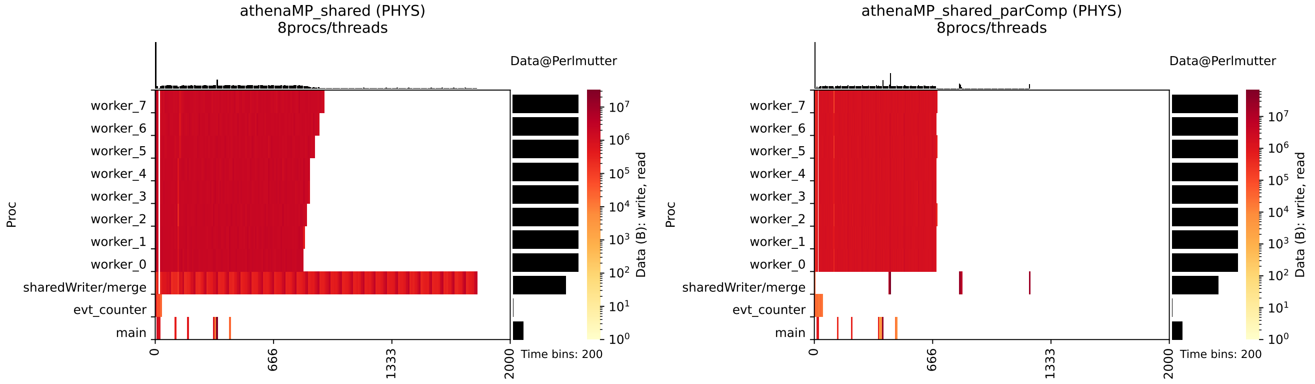
In this fourth use case we demonstrate how PyDarshan enables the analysis of large amounts of log files, which was prohibitively slow using the existing tools. This is important because Darshan is enabled by default at many facilities, and applications have been producing Darshan logs for years.

We first revisit Drishti from Section 3.2 but apply its analysis to very large numbers of Darshan logs from two different supercomputers. In a second example we discuss a more specialized investigation to quantify the amount of ROMIO MPI-IO coordination overhead. The two examples demonstrate how insights from sitewide analysis can be used to inform user training and improve storage systems as well as middleware designs.

3.4.1 Sitewide Analysis with Drishti to Understand Application Behavior. To inform the procurement or design of the next generation of storage systems, we need to understand and quantify how users and applications are using a system. For this use case we leverage Drishti's ability to flag I/O behaviors. Table 1 summarizes how frequently each insight was triggered in Cori (NERSC) and Theta (ANL) machines using a month's worth of Darshan data. We analyzed 517, 939 logs from Cori (April 2022) and 7, 595 from Theta (October 2022). Interesting trends can be observed in both machines. We highlight five examples from our analysis:

- High usage of the STDIO interface to transfer data (30% of the jobs in Cori and 45% in Theta).
- Low usage of collective I/O operations in both systems (triggered by 98% of the jobs in Cori and 85% in Theta).
- High number of small write/read requests using POSIX. In Cori, 62% of the jobs tracked with Darshan triggered this insight for read requests and 59% for write requests. In Theta, that represents 42% and 45% of the jobs, respectively.
- High number of misaligned memory and file accesses (66% of the jobs in Cori and 65% in Theta).
- High number of sequential write operations in both systems (66% in Cori and 56% of the jobs in Theta).

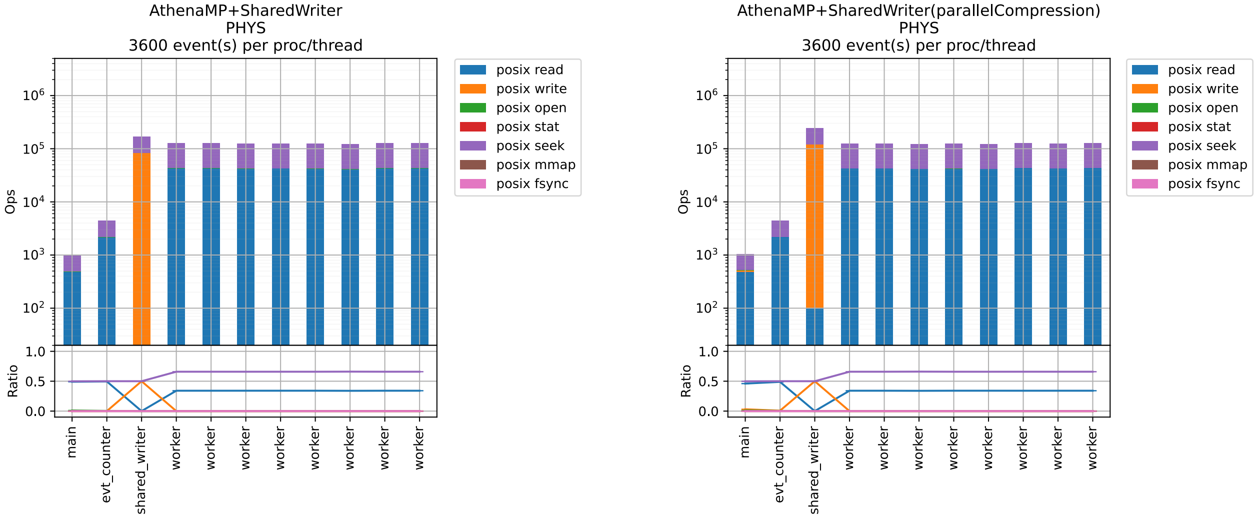
[5] https://home.cern/tags/lhc-run-2



(a) Shared Writer

(b) Shared Writer (Compression)

Figure 7: Heatmap of data read and written by an ATLAS Run3 Derivation production job, including the 8 workers and the single SharedWriter processes forked from the main AthenaMP process (left) and with parallel compression (right). For clarity, we only display access to input and output files.



(a) Shared Writer

(b) Shared Writer (Compression)

Figure 8: Number of POSIX operations (read, write, open, stat, seek, mmap, and fsync) in an ATLAS Run3 Derivation production job, including the 8 workers and the single SharedWriter processes forked from the main AthenaMP process. For clarity, we only display access to input and output files. The fraction of operations is shown in the bottom panels. The left one uses SharedWriter only. The right one enables the parallel compression along with the SharedWriter.

Such insights are helpful to both facilities and the research community in devising and implementing solutions to address those challenges (e.g., policies, optimization techniques, scheduling). Ather et al. [8] previously demonstrated Drishti's performance. When evaluated with hundreds of Darshan logs from Cori, Drishti took, on average, ≈ 10 seconds to generate each report.

3.4.2 Custom Sitewide Analysis to Investigate I/O Middleware. As a second example we perform large-scale log analysis using PyDarshan to quantify the coordination overheads in I/O middleware, for the ROMIO MPI-IO implementation. The workhorse optimization in ROMIO MPI-IO has been two-phase collective I/O [35]. Inside the ROMIO library, clients exchange data among themselves to transparently re-form the I/O into a more ideal workload. Two-phase collective buffering has two major costs: a data exchange cost and a coordination cost. Data exchange typically takes relatively little time on the high-speed networks of these machines. Coordination costs, when data exchange cannot occur before either the source or target of the exchange has arrived and is ready to participate in the collective, have so far proven challenging to quantify.

Table 1: Summary of insights triggered by Drishti in Cori (April 2022) and Theta (October 2022) Darshan instrumented jobs.

			Cori		Theta	
Level	Interface	Detected Behavior	Jobs	Total (%)	Jobs	Total (%)
HIGH	STDIO	High STDIO usage (> 10% of total transfer size uses STDIO)	158592	30.61	3430	45.16
INFO	POSIX	Write operation count intensive (> 10% more writes than reads)	75742	14.62	2695	35.48
INFO	POSIX	Read operation count intensive (> 10% more reads than writes)	267228	51.59	2282	30.04
INFO	POSIX	Write size intensive (> 10% more bytes written then read)	136664	26.38	1197	15.76
INFO	POSIX	Read size intensive (> 10% more bytes read then written)	207346	40.03	3201	42.14
HIGH	POSIX	High number of small (< $1MB$) read requests (> 10% of total read requests)	322130	62.19	3203	42.17
HIGH	POSIX	High number of small (< $1MB$) write requests (> 10% of total write requests)	309868	59.82	3386	44.58
HIGH	POSIX	High number of misaligned memory requests (> 10%)	191214	36.91	5012	65.99
HIGH	POSIX	High number of misaligned file requests (> 10%)	344503	66.51	4974	65.49
WARN	POSIX	Redundant reads	13113	2.53	605	7.96
WARN	POSIX	Redundant writes	1407	0.27	40	0.52
HIGH	POSIX	High number of random read requests (> 20%)	174148	33.62	738	9.71
OK	POSIX	High number of sequential read operations (≥ 80%)	169283	32.68	4273	56.26
HIGH	POSIX	High number of random write requests (> 20%)	2038	0.39	10	0.13
OK	POSIX	High number of sequential write operations (≥ 80%)	342581	66.14	4275	56.28
HIGH	POSIX	High number of small (< $1MB$) reads to shared-files (> 10% of total reads)	273274	52.76	2286	30.09
HIGH	POSIX	High number of small (< $1MB$) writes to shared-files (> 10% of total writes)	46384	8.95	1300	17.11
HIGH	POSIX	High metadata time (at least one rank spends > 30 seconds)	19443	3.75	364	4.79
HIGH	POSIX	Data transfer imbalance between ranks causing stragglers (> 15% difference)	286811	55.37	3136	41.29
HIGH	POSIX	Time imbalance between ranks causing stragglers (> 15% difference)	201206	38.84	2559	33.69
HIGH	POSIX	Write imbalance (> 30%) when accessing individual files	3178	0.61	578	7.61
HIGH	POSIX	Read imbalance (> 30%) when accessing individual files	5265	1.01	652	8.58
WARN	MPI-IO	No MPI-IO calls detected from Darshan logs	511731	98.80	6518	85.81
HIGH	MPI-IO	Detected MPI-IO but no collective read operation	3162	0.61	31	0.40
HIGH	MPI-IO	Detected MPI-IO but no collective write operation	612	0.11	1	0.013
OK	MPI-IO	Detected MPI-IO and collective read operations	1329	0.25	743	9.78
OK	MPI-IO	Detected MPI-IO and collective write operations	3266	0.63	1009	13.28
WARN	MPI-IO	Detected MPI-IO but no non-blocking read operations	6208	1.19	1077	14.18
WARN	MPI-IO	Detected MPI-IO but no non-blocking write operations	6208	1.19	1077	14.18

We developed a skew-indicative heuristic in an attempt to quantify coordination costs. Consider Figure 9. Some processes will not do any I/O due to I/O aggregation in the two-phase buffering algorithm, but all processes participate in data exchange. Should a process enter the collective late, as Rank 1 does, the other MPI processes will spend longer in the "exchange" phase waiting for the slow process to contribute data. Additionally, if the total amount of data to be read or written is larger than an internal buffer, the two-phase collective buffering algorithm will carry out this exchange and write process multiple times. Each round of this optimization cannot begin until the prior finishes. Because these I/O aggregators are writing to a shared resource, one process (such as Rank 0 in the figure) likely will spend a longer time in write than others—for example, because that process is stuck obtaining a lock from the underlying file system. This additional I/O time will delay other processes completing their exchange phase.

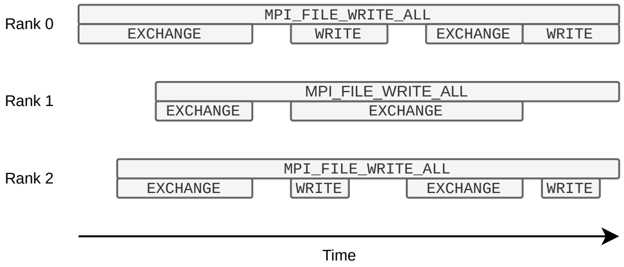


Figure 9: A collective MPI-IO call consists of data exchange followed by a write. If processes are not well synchronized, data exchange can take longer than expected. Skew can happen internally as well if one process spends longer in I/O than others.

In situations where all processes enter the collective simultaneously we would expect POSIX I/O time on these I/O aggregators to dominate the cost of a collective I/O write. Synthetic benchmarks behave in this way, for example. If processes are imbalanced or otherwise late in participating in the collective, we would expect to see that reflected in a high MPI-IO time. Rank 0 cannot proceed with writing until the data exchange with Rank 1 takes place.

Darshan logs can capture this discrepancy. Darshan collects a variety of statistics and timers from a program, including "slowest rank time" for both MPI-IO and POSIX. If the slowest MPI time is higher than the slowest POSIX time, that difference suggests higher time in the data exchange phase, which in turn suggests skewed processes.

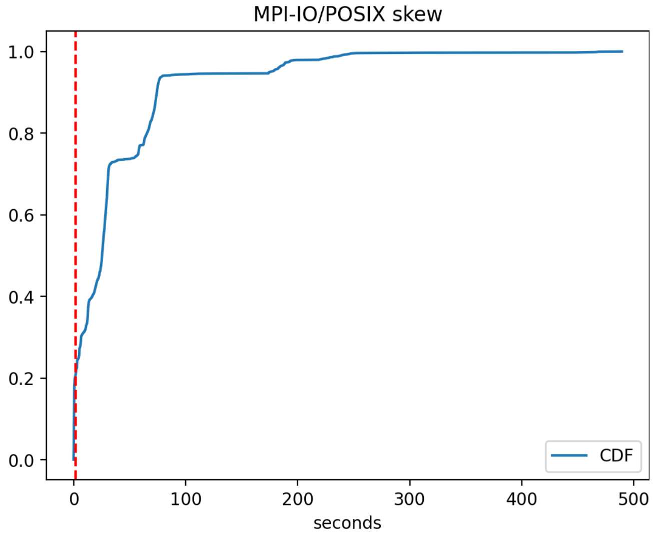


Figure 10: CDF plot depicting how often Darshan reported skew when writing to or reading from a file; for example, 21% of files reported a skew of two seconds or less.

We scanned all Darshan logs on the ALCF Theta machine for the arbitrarily chosen month of October 2020, applying our skew-indicative heuristic. In Fig. 10, we plot the cumulative distribution of our heuristic for the 377 applications that used MPI-IO to read or write at least one file. We find that on a per-file basis about 21% (458 out of 2,172 files accessed – programs operated on one or more files) exhibited skew of less than 2 seconds. One application reported over 400 seconds of skew. We expected to see some degree of skew, but this result surprised us. We plan on following up with these applications to better understand these skew results.

4 RELATED WORK

We group related work into three categories: (1) instrumentation solutions that provide their own ecosystem of analysis tools; (2) tools that strive for interoperability either by abstracting across different solutions or to build a bridge to advanced analysis such as data science or machine learning library; and (3) observability approaches popular in distributed-systems and cloud environments.

Various instrumentation solutions such as Score-P [26], Caliper [14], or Recorder [36] provide thin Python bindings to access the log data. Some of these do also capture I/O activity on different levels, such as POSIX and MPI-IO, similar to Darshan.

In order to interpret performance data, having additional context about the system and the software environment is often vital. Darshan attempts to capture some of this context for storage systems, for example, through the Darshan Lustre module. Other tools such as Adiak [5] introduce standardized system/application metadata capture that also could be helpful to augment Darshan log data.

The collection of performance analysis tools is not a new idea; tool suites exist for HPCToolkit [7, 28], Score-P [26], and TAU [33]. While customization is possible, there remains a relative high barrier of integration with custom analysis. This is why PyDarshan and other performance analysis tools [38] emphasize analysis in Jupyter Notebooks. Hatchet [13] and Thicket [15], for example, generalizes and develops prepared analysis and visualization for multidimensional performance data. Similar to PyDarshan, Thicket attempts to preserve data relationships but with a focus on comparing or aggregating from call trees across executions that can have arbitrary metrics attached.

Especially in cloud computing, standards for telemetry collection and analysis tools for distributed systems are emerging [4, 25, 29, 30]. As HPC and cloud technologies converge and containerization becomes more widely available, HPC centers are exploring leveraging the same technologies and tools. On the instrumentation side, the kernel-level eBPF [2, 21], promises additional control and granularity also useful for interception and counting of individual I/O calls. At the time of writing, however, we are unaware of a Darshan-like interface to standardized counters and data structures for I/O analysis, nor is it clear how kernel-level monitoring would be mapped down to individual application-specific reports.

5 CONCLUSION AND FUTURE WORK

In this work we introduced PyDarshan, a Python-based library to enable agile analysis of I/O performance data. PyDarshan caters to both novice and advanced users by offering ready-to-use HTML reports as well as different APIs to conduct custom analyses, for example, in Jupyter Notebooks. In four use cases we discuss how PyDarshan (1) improves single log analysis, (2) enables third-party developers to build new custom tools for I/O analysis on top of PyDarshan, (3) allows building I/O analysis tools for workflows, and (4) makes analysis of large quantities of jobs feasible because of more efficient access to Darshan log data.

In future work, we are looking to further optimize the underlying Darshan log data representation to improve analysis performance. The current Darshan binary log format and low-level utility library have two key drawbacks. The Python bindings must currently load a single record at a time into memory (thus introducing excessive function call and language binding overhead), and there is no a priori indication of the number records stored in a binary log file (making it difficult to plan an efficient memory layout up front).

We are considering several options to mitigate these problems. One option is to explore the feasibility of converting Darshan logs into Parquet files [32], which may allow us to more directly read in rectangular DataFrames through an Arrow-like memory representation. For example, Pandas [31, 37] has functions for directly reading parquet files into DataFrames.

Also of interest, in the more distant future, is the adoption of the in-development DataFrame API standard [6], so that we may allow consumers of PyDarshan to interchange the DataFrame framework in use. This may allow usage of the Polars library[6] in place of Pandas when parallel out-of-core analyses are required, the Dask library [19, 27] if distributing DataFrames over a cluster is desired, or cuDF [3] for GPU-based analyses of PyDarshan record DataFrames.

ACKNOWLEDGMENTS

This research used resources of the Argonne Leadership Computing Facility at Argonne National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under

[6] https://data-apis.org/dataframe-api/draft/

contract DE-AC02-06CH11357. This research was supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration. This research was also supported by the U.S. Department of Energy (DOE), Office of Science, Office of Advanced Scientific Computing Research (ASCR) under contract number DE-AC02-05CH11231 with LBNL. This research used resources of the National Energy Research Scientific Computing Center under Contract No. DE-AC02-05CH11231.

REFERENCES

[1] 2021. *The ATLAS Collaboration Software and Firmware.* Technical Report ATL-SOFT-PUB-2021-001. CERN, Geneva. ATL-SOFT-PUB-2021-001ATL-SOFT-PUB-2021-001

[2] 2023. BPF Compiler Collection (BCC). IO Visor Project. https://github.com/iovisor/bcc

[3] 2023. cuDF - GPU DataFrames. RAPIDS. https://github.com/rapidsai/cudf

[4] 2023. Grafana: The Open and Composable Observability and Data Visualization Platform. Grafana Labs. https://github.com/grafana/grafana

[5] 2023. LLNL/Adiak. Lawrence Livermore National Laboratory. https://github.com/LLNL/Adiak

[6] 2023. Pola-Rs/Polars. pola-rs. https://github.com/pola-rs/polars

[7] Laksono Adhianto, Sinchan Banerjee, Mike Fagan, Mark Krentel, Gabriel Marin, John Mellor-Crummey, and Nathan R Tallent. 2010. HPCToolkit: Tools for Performance Analysis of Optimized Parallel Programs. *Concurrency and Computation: Practice and Experience* 22, 6 (2010), 685–701.

[8] Hammad Ather, Jean Luca Bez, Boyana Norris, and Suren Byna. 2023. Illuminating The I/O Optimization Path Of Scientific Applications. In *High Performance Computing: 38th International Conference, ISC High Performance 2023, Hamburg, Germany, May 21–25, 2023, Proceedings*. Springer-Verlag, Berlin, Heidelberg, 22–41. https://doi.org/10.1007/978-3-031-32041-5_2

[9] ATLAS Collaboration. 2008. The ATLAS Experiment at the CERN Large Hadron Collider. *JINST* 3 (2008), S08003. https://doi.org/10.1088/1748-0221/3/08/S08003

[10] Jean Luca Bez, Hammad Ather, and Suren Byna. 2022. Drishti: Guiding End-Users in the I/O Optimization Journey. In *2022 IEEE/ACM International Parallel Data Systems Workshop (PDSW)*. 1–6. https://doi.org/10.1109/PDSW56643.2022.00006

[11] Jean Luca Bez, Houjun Tang, Bing Xie, David Williams-Young, Rob Latham, Rob Ross, Sarp Oral, and Suren Byna. 2021. I/O Bottleneck Detection and Tuning: Connecting the Dots Using Interactive Log Analysis. In *2021 IEEE/ACM Sixth Int. Parallel Data Systems Workshop (PDSW)*. 15–22. https://doi.org/10.1109/PDSW54622.2021.00008

[12] Shishir Bharathi, Ann Chervenak, Ewa Deelman, Gaurang Mehta, Mei-Hui Su, and Karan Vahi. 2008. Characterization of Scientific Workflows. In *2008 Third Workshop on Workflows in Support of Large-Scale Science*. IEEE, Austin, TX, USA, 1–10. https://doi.org/10.1109/WORKS.2008.4723958

[13] Abhinav Bhatele, Stephanie Brink, and Todd Gamblin. 2019. Hatchet: Pruning the Overgrowth in Parallel Profiles. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, Denver Colorado, 1–21. https://doi.org/10.1145/3295500.3356219

[14] David Boehme, Todd Gamblin, David Beckingsale, Peer-Timo Bremer, Alfredo Gimenez, Matthew LeGendre, Olga Pearce, and Martin Schulz. 2016. Caliper: Performance Introspection for HPC Software Stacks. In *SC '16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 550–560. https://doi.org/10.1109/SC.2016.46

[15] Stephanie Brink, Michael McKinsey, David Boehme, Connor Scully-Allison, Ian Lumsden, Daryl Hawkins, Treece Burgess, Katherine E Isaacs, Vanessa Lama, Michela Taufer, Jakob Luettgau, and Olga Pearce. 2023. Thicket: Seeing the Performance Experiment Forest for the Individual Run Trees. (2023).

[16] Rene Brun, Fons Rademakers, Philippe Canal, Axel Naumann, Olivier Couet, Lorenzo Moneta, Vassil Vassilev, Sergey Linev, Danilo Piparo, Gerardo GANIS, Bertrand Bellenot, Enrico Guiraud, Guilherme Amadio, wverkerke, Pere Mato, TimurP, Matevž Tadel, wlav, Enric Tejedor, Jakob Blomer, Andrei Gheata, Stephan Hageboeck, Stefan Roiser, marsupial, Stefan Wunsch, Oksana Shadura, Anirudha Bose, CristinaCristescu, Xavier Valls, and Raphael Isemann. 2019. Root-Project/Root: V6.18/02. Zenodo. https://doi.org/10.5281/zenodo.3895860

[17] Paolo Calafiura, Charles Leggett, Rolf Seuster, Vakhtang Tsulaia, Peter Van Gemmeren, and on behalf of the ATLAS Collaboration. 2015. Running ATLAS Workloads within Massively Parallel Distributed Applications Using Athena Multi-Process Framework (AthenaMP). *Journal of Physics: Conference Series* 664, 7 (Dec. 2015), 072050. https://doi.org/10.1088/1742-6596/664/7/072050

[18] Philip Carns, Kevin Harms, William Allcock, and Charles Bacon. 2011. Understanding and Improving Computational Science Storage Access through Continuous Characterization. *ACM Transactions on Storage* (2011), 25.

[19] Dask Development Team. 2016. *Dask: Library for Dynamic Task Scheduling.* https://dask.org

[20] DOE E3SM Project. 2021. Energy Exascale Earth System Model v2.0. [Computer Software] https://doi.org/10.11578/E3SM/dc.20210927.1. https://doi.org/10.11578/E3SM/dc.20210927.1

[21] eBPF authors. 2023. eBPF Website - Dynamically Program the Kernel for Efficient Networking, Observability, Tracing, and Security. https://ebpf.io

[22] Rafael Ferreira da Silva, Rosa Filgueira, Ilia Pietri, Ming Jiang, Rizos Sakellariou, and Ewa Deelman. 2017. A Characterization of Workflow Management Systems for Extreme-Scale Applications. *Future Generation Computer Systems* 75 (Oct. 2017), 228–238. https://doi.org/10.1016/j.future.2017.02.026

[23] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. 2020. Array programming with NumPy. *Nature* 585, 7825 (Sept. 2020), 357–362. https://doi.org/10.1038/s41586-020-2649-2

[24] S. Hoyer and J. Hamman. 2016. xarray: N-D labeled arrays and datasets in Python. *in prep, J. Open Res. Software* (2016).

[25] jaegertracing.io. 2019. Jaeger: Open Source, End-to-End Distributed Tracing. https://www.jaegertracing.io/

[26] Andreas Knüpfer, Christian Rössel, Dieter an Mey, Scott Biersdorff, Kai Diethelm, Dominic Eschweiler, Markus Geimer, Michael Gerndt, Daniel Lorenz, Allen D. Malony, Wolfgang E. Nagel, Yury Oleynik, Peter Philippen, Pavel Saviankou, Dirk Schmidl, Sameer Shende, Ronny Tschüter, Michael Wagner, Bert Wesarg, and Felix Wolf. 2011. Score-p: A Joint Performance Measurement Run-Time Infrastructure for Periscope, Scalasca, TAU, and Vampir. In *Tools for High Performance Computing 2011 - Proceedings of the 5th International Workshop on Parallel Tools for High Performance Computing, ZIH, Dresden, September 2011*, Holger Brunst, Matthias S. Müller, Wolfgang E. Nagel, and Michael M. Resch (Eds.). Springer. https://doi.org/10.1007/978-3-642-31476-6_7

[27] Matthew Rocklin. 2015. Dask: Parallel Computation with Blocked Algorithms and Task Scheduling. In *Proceedings of the 14th Python in Science Conference*, Kathryn Huff and James Bergstra (Eds.). 130–136.

[28] John Mellor-Crummey. 2003. HPCToolkit: Multi-platform Tools for Profile-Based Performance Analysis. In *5th International Workshop on Automatic Performance Analysis (APART)*.

[29] OpenTelemetry. 2019. OpenTelemetry: High-quality, Ubiquitous, and Portable Telemetry to Enable Effective Observability. https://opentelemetry.io/

[30] OpenZipkin. 2015. OpenZipkin: A Distributed Tracing System. https://zipkin.io/

[31] The pandas development team. 2020. Pandas-Dev/Pandas: Pandas. Zenodo. https://doi.org/10.5281/zenodo.3509134

[32] Parquet Contributers. 2023. Apache Parquet Documentation. https://parquet.apache.org/docs/

[33] Sameer S. Shende and Allen D. Malony. 2006. The Tau Parallel Performance System. *International Journal of High Performance Computing Applications* 20, 2 (May 2006), 287–311. https://doi.org/10.1177/1094342006064482

[34] Houjun Tang, Quincey Koziol, John Ravi, and Suren Byna. 2022. Transparent Asynchronous Parallel I/O Using Background Threads. *IEEE TPDS* 33, 4 (2022), 891–902. https://doi.org/10.1109/TPDS.2021.3090322

[35] Rajeev Thakur and Alok Choudhary. 1995. *Accessing Sections of Out-of-Core Arrays Using an Extended Two-Phase Method.* Technical Report SCCS-685. NPAC, Syracuse University.

[36] 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)*. 1–8. https://doi.org/10.1109/IPDPSW50202.2020.00176

[37] Wes McKinney. 2010. Data Structures for Statistical Computing in Python. In *Proceedings of the 9th Python in Science Conference*, Stéfan van der Walt and Jarrod Millman (Eds.). 56–61. https://doi.org/10.25080/Majora-92bf1922-00a

[38] Katy Williams, Alex Bigelow, and Katherine Isaacs. 2019. Visualizing a Moving Target: A Design Study on Task Parallel Programs in the Presence of Evolving Data and Concerns. *IEEE transactions on visualization and computer graphics* PP (Aug. 2019). https://doi.org/10.1109/TVCG.2019.2934285

[39] Cong Xu, Shane Snyder, Omkar Kulkarni, Vishwanath Venkatesan, Philip Carns, Suren Byna, Robert Sisneros, and Kalyana Chadalavada. 2017. DXT: Darshan eXtended Tracing. (2017).

[40] Weiqun Zhang, Andrew Myers, Kevin Gott, Ann Almgren, and John Bell. 2021. AMReX: Block-structured Adaptive Mesh Refinement for Multiphysics Applications. *The International Journal of High Performance Computing Applications* 35, 6 (2021), 508–526. https://doi.org/10.1177/10943420211022811