LDMS Darshan Connector: For Run Time Diagnosis of HPC Application I/O Performance

Sara Walton[1], Omar Aaziz[1], Ana Solorzan[2], and Ben Schwaller[1]

[1]Sandia National Laboratories, Albuquerque, USA
[2]Northeastern University, Northeastern University

***Abstract*—Periodic capture of comprehensive, usable I/O data for scientific applications requires an easy-to-use technique to record information throughout the execution without causing substantial performance effects. It will facilitate capturing the high variations of I/O performance and behavior that create a lack of understanding about the HPC application and help identify which components affect I/O variability and behavior. In this paper, we introduce a unique framework that provides low latency monitoring of I/O event data during run time. We implement a system-level infrastructure that continuously collects I/O application data from an existing I/O characterization tool to provide insights into the I/O application behavior and the components affecting it through analyses and visualizations. In this effort, we evaluate our framework by analyzing sampled I/O data captured from three HPC benchmark applications to understand the I/O behavior during the execution life of the applications. The result shows the success of capturing I/O patterns with acceptable overhead.**

I. Introduction

As more scientific applications are developed and used, the need for improved fidelity and throughput is more pressing than ever and much design effort and investments are being put into improving not only the application but also the system components. Being able to identify, predict and analyze I/O behaviors are critical to ensuring parallel storage systems are utilized efficiently. However, I/O performance continues to show high variations on large-scale production conditions in many cases [1]

Variations in I/O can be caused by the system usage (e.g. systems shared among users, time of the day being used), and by the system behavior such as the file system (e.g. buffering, file transfer, interrupt handling), network congestion, system resource contentions, or by the access patterns of the application itself. This variation makes it difficult to determine the root cause of I/O related problems and have a thorough understanding of throughput for system-specific behaviors and I/O performance in similar applications across a system. Further, not knowing the origin of such variations will directly affect the user and developer as unwanted time, effort and investment will need to be put into solving the issue.

Generally, the I/O performance is analyzed post-run by application developers, researchers and users in the form of regression testing or other I/O characterization tools that capture the applications I/O behavior. An example of one of these tools is *Darshan*, which monitors and captures I/O information on access patterns from HPC applications [2]. Efforts to identify the origin of I/O performance given by these tools usually come from any identified correlation between analyses of various applications runs or the time in which these applications were tested. However, this approach does not provide the ability to know **when** an I/O performance variability occurs during an application run and, if the developer or user wishes to, identify any correlations between the file system, network congestion or resource contentions and the I/O performance.

Execution logs that provide *absolute timestamps* (e.g. time-series) enable users and developers to perform temporal performance analysis, and better understand how these changing components affect the I/O performance variation as well as provide further insight into the application I/O pattern. Therefore, we introduce our *Darshan LDMS Integration* approach that incorporates the *absolute timestamps* to provide a run time set of application I/O data. This is a work in progress paper and will cover the following goals:

- Describe the approach used to expose absolute timestamp data from an existing I/O characterization tool;
- Provide a high level overview of the implementation process and other tools used to collect application I/O data during run time;
- Demonstrate use cases of the *Darshan-LDMS Connector* for four applications with distinct I/O behavior on a production HPC system;
- Utilize Darshan LDMS data to identify and better understand any root cause(s) of application I/O performance variation run time;
- Present how this new approach can be integrated with other tools to benefits users to collect and assist in the detection of application I/O performance variances across multiple applications.

Section II presents the background and motivation for this paper, Section III presents the approach to design the Darshan LDMS Integration and collect the absolute timestamps, Section IV depicts the tools integrated to our approach , Section V presents the experimental methodology to generated our results (Section VI). Section VII presents the related works, and Section VIII presents the conclusion and future works.

II. Background

Darshan is a lightweight I/O characterization tool that captures I/O access pattern information from HPC applica-

tions. [2] This will be described in detail in Section III. This I/O characterization tool will be used for tracing and collecting detailed I/O event data.

The Lightweight Distributed Metric Service (LDMS) is a low-overhead system that collects and transports HPC data for OVIS via *samplers* and *plugins*. [3] OVIS which is a modular system for collecting, analyzing, storing, transporting and visualization HPC data in order to provide further insights into the system state, resource utilization and performance. A *sampler* is a type of daemon that collects the data while the *plugin* determines the type of data to be sampled, aggregated or stored and a *sampler* is the set configuration for collecting the data specified by the plugin. There are a variety of samplers and other plugins that can be written for the LDMS API (in C). The system state insights are achieved by LDMS's *absolute-timestamp* view of system conditions through multi-hop *aggregation* and the LDMS Transport. An LDMS daemon, *LDMSD*, aggregator supports multiple levels, networks and security domains so data can be sent to various locations. Additional functionalities, such as the *LDMS Streams API*, has been developed to allow for aggregation of event-based application data. This framework utilizes *LDMS Streams API* to collect Darshan's I/O event data during execution time and store the *timestamped* data to a database.

The Distributed Scalable Object Store (DSOS) is a storage database designed to efficiently manage large volumes of HPC data [4]. It supports high data injection rates, has an enhanced query performance and flexible storage management. DSOS has a command line interface for data interaction and has various program API's such as Python, C and C++. A DSOS cluster consists of multiple instances of DSOS daemons, *dsosd*, that run on multiple storage servers on a single cluster. The DSOS Client API can perform parallel queries to all *dsosd* in a DSOS cluster. The results of the queried data re then returned in parallel and sorted based on the index selected by the user. This database and it's Python API are used in this framework for storing and querying the I/O event data.

The HPC Web Services is an analysis and visualization infrastructure [5], that integrates an open-source web application, Grafana [6] with python modules for analysis and visualization of HPC data. Grafana is an open-source visualization tool for various data sources and has a front-end interface website, Grafana dashbaord, that provides charts, graphs, tables, etc. for viewing and analyzing queried data in real time. The python analysis modules are specified in a Grafana dashboard where the queries are performed. Once specified, that python analysis transforms any data queried from the dashboard in real time. Data can then be filtered and viewed in differerent time-frames and formate in the Grafana dashboard as well as be saved and shared to other users who have access to the webpage. This framework will leverage the HPC Web Services for run time analyses and visualizations of the I/O event data.

III. Darshan LDMS Integration

This section provides a high-level overview of the design and implementation of the *Darshan LDMS Integration* and the components used to create this infrastructure. **Might comment out to the end of the enumerate list to remove the "fluff"** This approach will provide run time insights about application I/O by using the following tools:

1) The I/O characterization tool, Darshan, to collect application I/O behavior and patterns. However, this tool does not report the *absolute timestamp* so modifications to the code were made to expose this data.
2) LDMS to provide and transport live run time data feed about application I/O events. [7]
3) DSOS to store and query large amounts of data generated on a production HPC system. [4]
4) HPC Web Services to present run time I/O data. The timeseries data will enable the user to identify when a variability occurs as well as create new meaningful analyses. [5]

Darshan is used to tune applications for increased scientific productivity or performance and is suitable for full time deployment for workload characterization of large systems [8]. It provides detailed statistics about various level file accesses made by MPI and non-MPI applications which can be enabled or disabled as desired. These levels include POSIX, STDIO, LUSTRE and MDHIM for non-MPI applications and MPI-IO, HDF5 and some PnetCDF for MPI applications [9]. This functionality provides users with a summary of I/O behavior and patterns from an application run but it does so post-run. Therefore, it does not allow insights into *run time* I/O behavior and patterns which makes it nearly impossible to identify the root cause(s) of I/O variability and when this occurs.

LDMS is used to efficiently collect and transport scalable *synchronous* and *event-based* data with low-overhead. Two key functionalities it has that will be leveraged in the *Darshan LDMS Integration* and create the *Darshan-LDMS Connector* are the *LDMS Streams* and transport [10]. In the *Darshan LDMS Integration* , the LDMS was enhanced to support application I/O data injection and store to DSOS while Darshan was modified to expose the *absolute timestamp* and publish run time I/O events for each rank to the *LDMS Streams*. This integration will be described in detail later on.

DSOS will enable the ability to query the timestamped application I/O data through a variety of APIs while Grafana will provide a front-end interface for visualizing the stored data that has been queried and analyzed using Python based modules on the back-end. With these tools, users can view, edit and share analyses of the data as well as create new meaningful analyses.

The implementation of LDMS into Darshan along with the storage, analysis and visualization components that make up this design will provide detailed insights into the I/O behavior and patterns during run time. This insight will allow users and researchers to better understand how application I/O variability correlates with overall system behavior (e.g. file

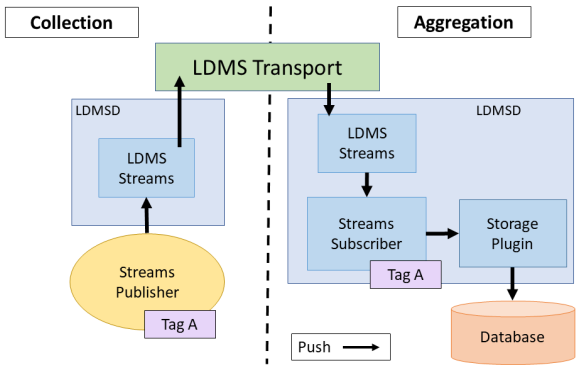

Fig. 1: Overview of the LDMS event data collection. Application data is *pushed* by publishing data to the *LDMS Streams* which is then *pushed* through the LDMS transport to the *LDMS Streams* LDMS aggregator (right) where the data is *pushed* to the streams subscriber (Tag A) and stored to a database.

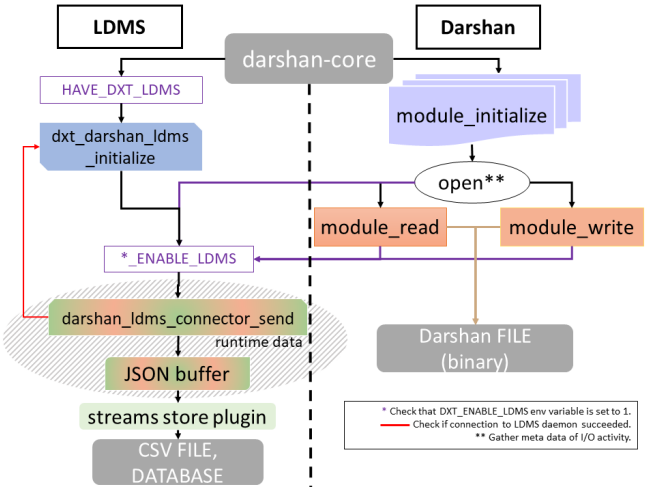

Fig. 2: Overview of the *Darshan-LDMS Connector* design and how it collects I/O data for each read, write, open and close events per rank from Darshan. The LDMS library must be linked against the Darshan build in order to utilize the *LDMS Streams* functionality and store plugins.

system, network congestion, etc.) how the time of day affects the I/O performance, how the I/O pattern within a run affects the I/O performance of read and write and why the read and write I/O performance patterns are different and independent of each other. Further, the occurrence of any I/O variability can be identified during run time.

IV. Integration Tools

A. Darshan

Darshan is divided into two main parts: 1) *darshan-runtime* which contains the instrumentation of the characterization tool and produces a single log file a the end of each execution summarizing the I/O access patterns used by the application [11]. 2) *darshan-util* which is intended for analyzing log files produced by darshan-runtime [11]. The *Darshan LDMS Integration* focuses on the *darshan-runtime* [9] as this is where the source code of I/O event data is recorded by Darshan.

Darshan tracks the start, duration and end time of an application run via the C function *clock_gettime()* and converts the result into seconds and passes the result to a struct that is then used to report the summary log files [12]. Therefore, in order to retrieve the *absolute timestamp* and include it into the I/O event data during run time, a time struct pointer was added to the function call that used *clock_gettime()* in *darshan-runtime*. This pointer was passed through all of Darshan's modules and the *absolute timestamp* was collected. This was the preferred method as it required minimal changes to Darshan's source code and no additional overhead and latency between the function call and recording of the *absolute timestamp*.

B. LDMS Streams

The word, *LDMSD*, refers to an LDMS daemon that provides the capability of data collection, transport and storage and their *plugins* determine the functionality of these capabilities [7]. Daemons on the compute nodes run sampler plugins and transport is achieved through multi-hop *aggregation*. LDMS had two levels of aggregator daemons [7]which can utilize storage plugins to store any sets of data into various formats so long as it's specified beforehand.

The *Darshan LDMS Integration* leverages the LDMS transport to support the injection and transport of application I/O data which requires a *push-based* method to reduce the amount of memory consumed and data loss on the node as well as reduce the latency between the time in which the event occurs and when it is recorded. A *pull-based* method would require a buffering to hold an unknown number of events between pulls. Also, the transported data format needs to support *variable-length* events because the I/O data may will most likely vary in size.

This leads to the LDMS *publish-subscribe bus* capability , *LDMS Streams*, which has been enhanced to support I/O event data. This capability is intended for publishing and subscribing to an *LDMS streams tag*. The tag needs to be specified in LDMS daemons and *plugins* in order to publish event data to *LDMS Streams* and receive this published *LDMS Streams* data that match the tag.This process and the *push-based* method can be seen in Figure 1. Event data can be specified as either `string` or `JSON` format. The *LDMS Streams* API was modified to support long application connections and message injections. *LDMS Streams* uses best effort without a reconnect or resend for delivery and does not cache it's data so the published data can only be received after subscription. The *LDMS Streams* allows the ability for any data source to be injected into the LDMS transport.

C. Darshan Connector

The most recent version of Darshan allows for full tracing of application I/O workloads using their DXT instrumentation module which can be enabled and disabled as desired at

JSON Message

{ "uid":99066,"exe":"<absolute-path>/mpi-io-test","job_id":226903,"rank":3,"ProducerName":"nid00046","file":"<absolute-path>/mpi-io-test.tmp.dat","record_id":160154300648089062,"module":"POSIX","type":"MET","max_byte":-1,"switches":-1,"flushes":-1,"cnt":1,"op":"open","seg":[{"data_set":"N/A","pt_sel":-1,"irreg_hslab":-1,"reg_hslab":-1,"ndims":-1,"npoints":-1,"off":-1,"len":-1,"dur":0.002069,"timestamp":1653416252.990068}]}

{"uid":99066,"exe":"N/A","job_id":226901,"rank":2,"ProducerName":"nid00046","file":"N/A","record_id":160154300648089062,"module":"POSIX","type":"MOD","max_byte":-1,"switches":-1,"flushes":-1,"cnt":1,"op":"close","seg":[{"data_set":"N/A","pt_sel":-1,"irreg_hslab":-1,"reg_hslab":-1,"ndims":-1,"npoints":-1,"off":-1,"len":-1,"dur":0.002296,"timestamp":1653416253.006978}]}

CSV Header

#module,uid,ProducerName,switches,file,rank,flushes,record_id,exe,max_byte,type,job_id,op,cnt,seg:off,seg:pt_sel,seg:dur,seg:len,seg:ndims,seg:reg_hslab,seg:irreg_hslab,seg:data_set,seg:npoints,seg:timestamp

Fig. 3: Output of a MPI-IO Darshan test run in the JSON format (top image), and the CSV file header (bottom). The JSON message is published to the *LDMS Streams* interface where it is then converted to CSV and stored to *DSOS*. The `name:value` pairs in light blue indicate meta data stored, while the light purple indicates the file level access data not applicable to POSIX and are given the values of `"N/A"` or `"-1"`. The `"seg"` is a list containing multiple `name:value` pairs.

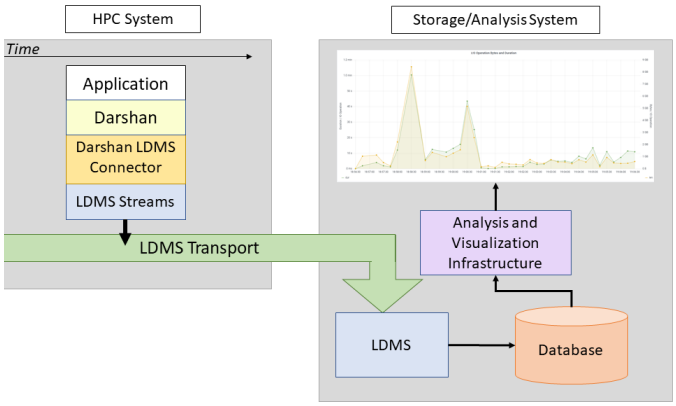

Fig. 4: Overview of the *Darshan LDMS Integration* where the *Darshan-LDMS Connector* is used to intercept the I/O behavior Darshan is collecting utilizes the various tools to publish, store, analyze and view runtime I/O behavior.

runtime. DXT provides high-fidelity traces for an application's I/O workload vs Darshan's traditional I/O summary data and currently traces POSIX and MPI-IO layers [9]. This design leverages the additional I/O tracing Darshan's DXT provides through the new *Darshan-LDMS Connector* capability.

The *Darshan-LDMS Connector* functionality collects both DXT data and Darshan's original I/O data and optionally publishes a message in JSON format to the *LDMS Streams* interface as seen in Figure 3. The *absolute timestamp* is also included in this message with the given name "timestamp". The LDMS transport then transports the I/O event data to a *DSOS* database where Grafana can access and query this data. the *Darshan-LDMS Connector* currently uses a single unique *LDMS Stream tag* for this data source. For the file level accesses that DXT does not trace or for file level access type that have different name-value pairs, a value of "N/A" or "-1" is given in the JSON message.

Darshan has a large number of metrics it uses for I/O tracing and post-processing calculations. The current stages of this framework collects a subset of these metrics to publish to *LDMS Streams*, as presented in Figure 3. These metrics provide the most value to the user as they will provide the ability to create new I/O behavior analyses and visualizations to get further insights of the application I/O behavior, and reveal correlations between I/O performance variability and system behavior. Table I depicts the names and definition of each metric in the JSON file. Depending on the `"type"` input, the absolute directory of the Darshan file output and executable will be recorded and published to *LDMS Streams*. If `"type"` is set to `"MET"` (e.g. "meta"), the absolute directories will be recorded. Otherwise, it will receive the value "N/A" if set to `"MOD"`(e.g. "module"). The `"type"` will be set to "MET" for open I/O events, which are the Darshan I/O records that have permanent values during the application execution, such as the rank, file and node name. The `"type"` is set to "MOD" for all other I/O events to reduce the message size and latency when sending the data through an HPC production system pipeline.

D. Storage: DSOS Database

DSOS is built on the Scalable Object Store (SOS) database [4] and was intended to address the domain-specific needs of large-scale HPC monitoring. It was chosen as the preferred database because it allows for interaction via a command line interface which allows for fast query testing and data examination. DSOS also provides scalable data ingest and the ability to query large volumes of data which is required for the large amounts of data to be ingested and stored. To sort though the published *LDMS Streams* data, combinations of the job ID, rank and timestamp are used to create joint indices where each index provided a different query performance. An example of this is using `job_rank_time` which will order the data by job, rank then timestamp and then search the data by a specific rank within a specific job over time.

E. Analysis and Visualization: HPC Web Services

The HPC Web Services [5] is an infrastructure that consists of the analysis and visualization components of this approach. Any data queries start from a front-end application and transferred to a back-end application that are running on an HPC cluster. In this case the front-end website is Grafana [6] and the back-end consists of Python analysis modules. The HPC Web Services also provide instant analysis where data can be analyzed and viewed in real time as opposed to the traditional method of querying the results of analyzed data from a separate database.

Grafana is an open-source visualization application that provides various charts, graphs and alerts for supported data sources. It can support multiple data formats but is best suited for timeseries data. It has storage plugins for many database technologies in order to query and render data from multiple

uuid	User ID of the job run
exe	Absolute directory of the application executable
module	Name of the Darshan module data being collected
ProducerName	Name of the compute node the application is running on
switches	Number of times access alternated between read and write
file	Absolute directory of the filename where the operations are performed
rank	Rank of the processes at I/O
flushes	Number of "flush" operations. It is the HDF5 file flush operations for H5F, and the dataset flush operations for H5
record_id	Darshan file record ID of the file the dataset belongs to
max_byte	Highest offset byte read and written per operation
type	The type of JSON data being published: MOD for gathering module data or MET for gathering static meta data
job_id	The Job ID of the application run
op	Type of operation being performed (i.e. read, write, open, close)
cnt	The count of the operations performed per module per rank. Resets to 0 after each "close" operation
seg	A list containing metrics names per operation per rank
seg:pt_sel	HDF5 number of different access selections
seg:dur	Duration of each operation performed for the given rank (i.e. a rank takes "X" time to perform a r/w/o/c operation)
seg:len	Number of bytes read/written per operation per rank
seg:ndims	HDF5 number of dimensions in dataset's dataspace
seg:reg_hslab	HDF5 number of regular hyperslabs
seg:irreg_hslab	HDF5 number of irregular hyperslabs
seg:data_set	HDF5 dataset name
seg:npoints	HDF5 number of points in dataset's dataspace
seg:timestamp	End time of given operation per rank (in epoch time)

TABLE I: Metrics defined in the JSON file published to the *Darshan LDMS Integration*.

data sources. The *Darshan LDMS Integration* implemented a storage plugin for the DSOS database in order to query this data and visualize it on the Grafana web interface [6]. An overview of the this integration can be seen in Figure 4.

Python analysis modules are used to produce meaningful visualizations on the queried data from the DSOS database. With these modules, queried data is converted into a pandas dataframe to allow for easier application of complex calculations, transformations and aggregations on the data. The type of analysis module is specified in the Grafana web interface. This is where the *Darshan LDMS Integration* will demonstrate how runtime I/O data will provide further insights and understanding into application I/O behavior, patterns, performance variability and any correlations these have with the system behavior.

V. Experimental Methodology

This section presents our experimental methodology to evaluate our framework using four applications with different I/O behavior: HACC-IO, HMMER applications and the Darshan MPI-IO benchmark. We performed the experiments in a Cray HPC cluster using NFS and Lustre file systems.

A. Applications

- HACC-IO is the I/O proxy for the large scientific Hardware Accelerated Cosmology Code (HACC), an N-body framework that simulates the evolution of mass in the universe with short and long-range interactions [13]. The long-range solvers implement an underlying 3D FFT. HACC-IO is an MPI code that simulates the POSIX, MPI collective, and MPI independent I/O patterns of fault tolerance HACC checkpoints. It takes a number of particles per rank as input, writes out a simulated checkpoint information into a file, and then read it for validation. We ran HACC-IO with several configurations to simulate different workloads on the NFS and Lustre file systems. Table IIb shows the different run configurations.
- HMMER is a suite of applications that profiles a hidden Markov model (HMM) to search similar protein sequences [14]. HMMER has a building code called "hmmbuild" that uses MPI to build a database by concatenating multiple profiles Stockholm alignment files. In our experiment, we used the Pfam-A.seed [15] file to generate a large Pfam-A.hmm database. We ran HMMER with 32 MPI ranks on one node, and we ran it in two configurations where we point the database file to NFS and then Lustre, respectively.
- MPI-IO-TEST benchmark is a Darshan utility that exists in the code distribution to test the MPI I/O performance on HPC machines. It can produce iterations of messages with different block sizes sent from various MPI ranks. It can also simulate collective and independent MPI I/O methods. We experimented with NFS vs. Lustre and collective vs. independent MPI I/O. We ran the benchmark with four configurations on 22 nodes and set the number of iterations to 10 and the block size to 16MB. Table IIa shows the different configuration used.
- sw4 is a geodynamics code that solves 3D seismic wave equations with local meshrefinement [16]. sw4 accepts an input file that specifies the 3D grid simulation size, and we selected a size that uses about 50% of the available memory to memic a realistic run of the application.

B. Evaluation System

We experiment using several I/O loads on the Voltrino Cray XC40 system at Sandia National Laboratories. The system

has 24 diskless nodes with Dual Intel Xeon Haswell E5-2698 v3 @ 2.30GHz 16 cores, 32 threads/socket, 64 GB DDR3-1866MHz memory, and connected with a Cray Aries DragonFly interconnect. The machine has two file systems: the network file system (NFS) and the Lustre file system (LFS).**[TODO1: File systems???]**

C. Enviroment

Voltrino, run LDMS samplers on the compute nodes and one LDMS aggregator on the head node. LDMS uses the UGNI interface to transfer Darshan streams data and other performance metrics from the compute nodes to the head node. The aggregator on the head node transmits the data to another LDMS aggregator on another cluster, Shirley, for analysis and storage. Shirley, host the HPC web services Grafana application and the DSOS database. Darshan can wrap the I/O function in an application by linking the executables staticky and dynamic. Our framework uses dynamic linking to collect darshan data. So we set the `LD_Preload` environment variable to point the path of the Darshan library shared objects which have the LDMS streams API calls to send the data through the LDMS compute node daemons.

VI. Results

This section covers what significance of the approach to collecting runtime application I/O data using *Darshan LDMS Integration* . We present performance analysis that shows how the new metrics helped provide more insight results into I/O behavior and how this information can be represented in Grafana.

A. Experiments and Overhead

Each application was tested on the Lustre and NFS file system with various configurations for each application run. **HMMER could only run on one node because...** All application experiments were repeated 5 times for the *Darshan-LDMS Connector* and Darshan only (i.e. no LDMS implemented) which summed to a total of 110 job submissions. The layout for these runs are shown in Table II.

Took another stab at this. FIX ANYTHING THAT DOESN'T MAKE SENSE. OR REMOVE IT ALL TOGETHER. The average of the 5 execution times (e.g. Average Runtime (s)) for Darshan and the *Darshan-LDMS Connector* (e.g. dC) was then taken and used to calculate the percent overhead of LDMS. The runtimes with Darshan only was performed and recorded 1-2 weeks before the experiments with the *Darshan-LDMS Connector* . As seen from Table IIa, the overhead of LDMS on Darshans' MPI-IO-TEST benchmark for three of the experiments shows an decrease in overall runtime with the *Darshan-LDMS Connector* . Since this is not feasible, the runtime improvement seen with the *Darshan-LDMS Connector* is most likely due to the NFS and Lustre file systems where these experiments were performed. Since the applications for Darshan and the *Darshan-LDMS Connector* were ran at different times, the file systems could have performed worse during the Darshan experiments and

MPI-IO-TEST				
File System	NFS		Lustre	
Nodes	22		22	
Block Size	16*1024*1024		16*1024*1024	
Iterations	10		10	
Collective	Yes	No	Yes	No
Avg. Messages	50390	6397	25770	15676
Rate (msgs/sec)	37	7	95	38
Average Runtime (s)				
Darshan	1376.67	880.46	249.97	428.18
dC	1355.35	858.68	270.98	414.35
% Overhead	-1.55%	-2.47%	8.41%	-3.23%

(a) MPI-IO

HACC-IO				
File System	NFS		Lustre	
Nodes	16		16	
Particles/Rank	5000000	10000000	5000000	10000000
Avg. Messages	1663	1774	1995	1711
Rate (msgs/sec)	2	1	3	2
Average Runtime (s)				
Darshan	882.46	1353.87	417.14	1616.87
dC	775.24	1365.24	467.24	1027.44
% Overhead	-12.15%	0.84%	12.01%	-36.45%

(b) HACC-IO

HMMER		
File System	NFS	Lustre
Nodes	1	1
Input	Pfam-A.seed	
Avg. Messages	3117342	4461738
Rate (msgs/sec)	1483	2396
Average Runtime (s)		
Darshan	749.88	135.40
dC	2826.01	1863.98
% Overhead	276.86%	1276.67%

(c) HMMER

TABLE II: Overview of each experiment configuration, target file system, average elapsed time (s) from 5 runs and calculated overhead of LDMS.

better during the *Darshan-LDMS Connector* . This change in file system performance would have a direct affect on the overall runtime and create a false calculation of increased overhead. We have not been able to test on a different cluster or have had the opportunity to interleave the experiments where each Darshan only run is followed by a *Darshan-LDMS Connector* run. These experiments will be part of further research and understanding as to why we are seeing a performance improvement. The other experiment on the Lustre file system with Collective enabled shows an overhead of less than 8.41% which is a minor overhead as this experiment has the highest rate of messages being sent per second to the LDMS Streams interface..

The HACC-IO application, seen in Table IIb was similar to the MPI-IO-TEST bench mark in regards to a shorter runtime with the *Darshan-LDMS Connector* for both file systems. Again, this is most likely due to the Lustre and NFS file systems as described earlier. The other two experiments, NFS

with 10million particles and Lustre with 5million particles, show an overhead of 0.84% and 12.01% respectively. The experiment with 0.84% overhead indicates there is no significant affect to the applications runtime while the experiment with 12.01% overhead does show a longer runtime with the *Darshan-LDMS Connector* but, similar to the MPI-IO-TEST benchmark, it has the highest message rate of 3msgs/sec.

The HMMER application, seen in Table IIb shows a significant increase in runtime with upwards of 270% for NFS and 12000% for Lustre where the user has to wait 2x-12X longer for their application to finish. This, of course, is not ideal for the average user and the result of these high overhead percentages are due to the json message formatting. Additional tests have been performed without the sprintf() function to generate the json message (i.e. only LDMS Streams API is enabled and the *Darshan-LDMS Connector* send function is called) and the average overhead was 0.37%. This indicates that the increase in overhead is not due to LDMS but rather the formatting of the I/O event data from integers into strings. HMMER had an average of 3-4 million messages (i.e. Darshan I/O events) during a single run with a rate of 1-2k messages per second. HACC-IO and MPI-IO-TEST, on the other hand, had an average of 1-50K messages with rates of less than 100 messages per second. This is significantly lower than HMMER which is why no significant overhead was detected.

Darshan collects all I/O events of an application run and the *Darshan-LDMS Connector* is implemented such that when Darshan detects an I/O event, the *Darshan-LDMS Connector* will collect and format that current set of I/O metrics into a json message. In order to send a json message, all integers must be converted to strings and this conversion comes at a performance cost. Therefore, the more I/O intensive an application is and the shorter the runtime, the overhead will increase significantly and cause the runtime performance to drop (as seen in HMMER). Since there is currently no other way to send I/O data as a json message to the LDMS Streams interface without converting the integers to strings, we must pay this performance cost. To overcome this issue, we plan to further develop the *Darshan LDMS Integration* framework to allow users to collect every *n-th* I/O event detected by Darshan. This functionality will allow users to analyze a percentage of their applications I/O behavior during runtime without having to compensate in runtime performance. **END OF THE SECTION I'VE ADDED - 6/26**

B. Analysis and Grafana Output

Figure 5 presents the mean number of I/O operations for each HACC application and the error bar considering 95% confidence interval for the five jobs. This plot shows that even running at the same system and with the same configuration, the applications presented different I/O behavior. In fact, a single HPC application can have multiple unique I/O behavior that can degradate the performance of the application [1]. This I/O variation can also happen between allocated devices. Figure 6 shows the number of I/O requests per node for close and open operations for two jobs for the HACC-IO application on Lustre for 10 millions particles per rank.

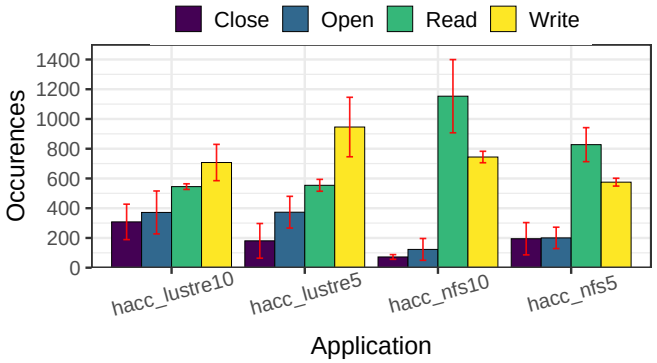

Fig. 5: The same application can perform different amount of I/O operations during execution. It shows the mean occurences of each operation over the five job runs.

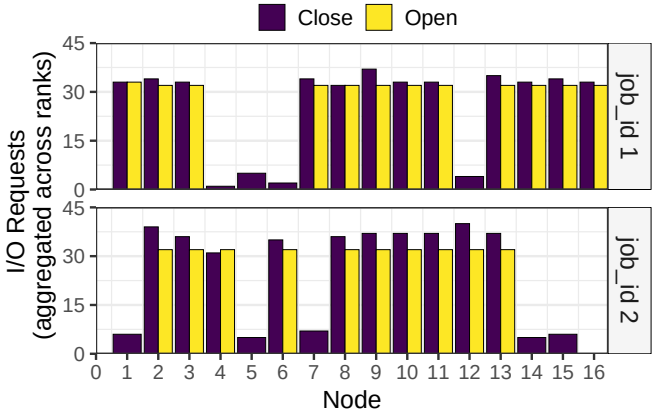

Fig. 6: The same application can perform different amount of I/O operations per node.

Figure 7 shows the duration of the reads and writes per rank for each execution (`job_id` metric) of the MPI-IO benchmark without using collective operations. We notice a similar behavior for the I/O operations duration for all jobs except the second one (`job_id 2`). It presents a mean duration of 6.75 seconds for reads and 78s for writes, while the other jobs had a mean duration of 0.05s for reads and 54s for writes. With the collected logs, we can perform a spatial performance analysis to understand the variability in the I/O behavior per system component, in this case, per nodes and ranks.

Using the absolute timestamps collected we can temporarily view wherein the application execution the variability of a job occured. Figure 8 presents the duration and occurence of I/O operations throughout the MPI-IO benchmark for `job_id 2`. We can identify the application I/O pattern of performing writings during ten phases, and then reads at the end. Also, this application run faster writes at the beginning and slower

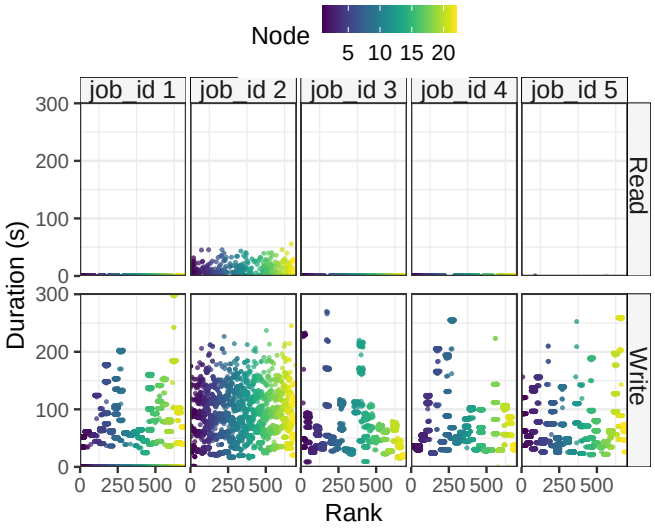

Fig. 7: Jobs for the MPI-IO benchmark without collective operations presented variability in the number and duration of I/O operations.

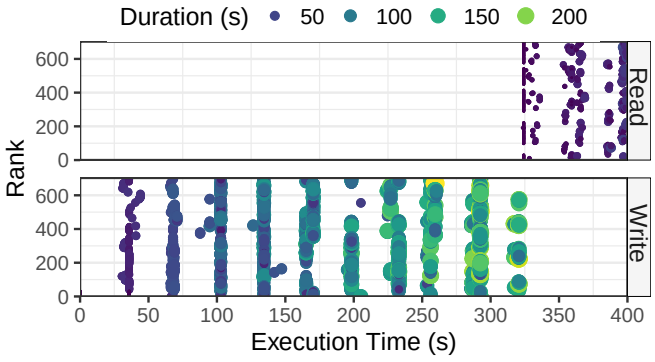

Fig. 8: Distribution of reads and writes operations throughout the execution time for the `job_id 2`, can reveal the application I/O pattern, and wherein the application there were faster and slower operations.

at the end, with the slowest writting after 250 seconds.

The same job is also represented in Figure 9 using the Grafana interface. This figure presents the number of I/O writes (blues) and reads (green) and the amount of bytes sent aggregated across ranks. The writes behavior shows the application phases where it dealt with larger I/O sizes, with two moments writting more than 20GB, while the reading operations run for a shorter moment for aroun 12GB of I/O size. Grafana offers an interactive front-end view where users can easily filter to visualize specific time and metrics intervals. Such representation using the absolute timestamps facilitates the correlation of I/O performance congestion, for example, with system behavior monitoring, which can also be represented as a Grafana dashboard.

VII. Related Work

Extensive research has already been proposed in literature to provide further insights into I/O behavior. The PASSION Runtime Library for parallel I/O proposed by Syracuse University [17] that optimizes I/O intensive applications through Data Prefetching and Data Sieving, the IOPin: Runtime Profiling of Parallel I/O in HPC Systems proposes a dynamic instrumentation to show the interactions from a parallel I/O application to the file system [18] and Design and Implementation of a Parallel I/O Runtime System for Irregular Applications [19] that proposes two different collective I/O techniques for improving I/O performance.

Darshan was the preferred I/O characterization tool because they had their Darshan's eXtended Tracing (DXT) [9] instrumentation module that provides high-fidelity traces for an application's I/O workload vs Darshan's traditional I/O summary data [9]. Other open-source I/O tools that we have come across or are aware of do not have this extensive I/O tracing capability which is leveraged in this work.

This work in progress differs from these approaches because we *leverage and enhance* existing applications and tools to design an infrastructure that creates runtime analyses and visualizations from detailed traces of application I/O events during execution time. The *Darshan LDMS Integration* integrates LDMS's *time stamped* data collection and storage capabilities [10] with Darshan [8] to collect runtime application I/O data. Further, a database is implemented to allow for efficient queries of large volumes of data as well as python analysis modules and an open-source web application for runtime analyses and visualizations.

VIII. Future Work

Taking a stab at this This paper covered the *Darshan LDMS Integration* design and implementation of the *Darshan-LDMS Connector* which collected I/O data from an I/O characterization tool to create a new timeseries that allows for further insights into I/O behavior and patterns. Five key components were used to develop this design which were the I/O event data (Darshan), data collection (LDMS Streams), storage (DSOS), analysis (Python modules) and visualization (Grafana). These results of this design proved to enhance both LDMS and Darshan tools as well as create new insights and provide a better understanding to application I/O performance and behavior.

The next steps are to further expand the *Darshan-LDMS Connector* and it's capabilities by including more I/O event data and demonstrating advanced insights into correlations between I/O performance and system behavior.However, an issue with increased overhead does arise when collecting data from short but intensive I/O applications (i.e. Hmmer). This is because the I/O event data consists of multiple integers that must be formatted into a json message and this integer to string conversion comes at a performance cost. The *Darshan-LDMS Connector* is currently implemented to collect and format the I/O data whenever Darshan detects an I/O event. Darshan collects every I/O event that occurs during an application

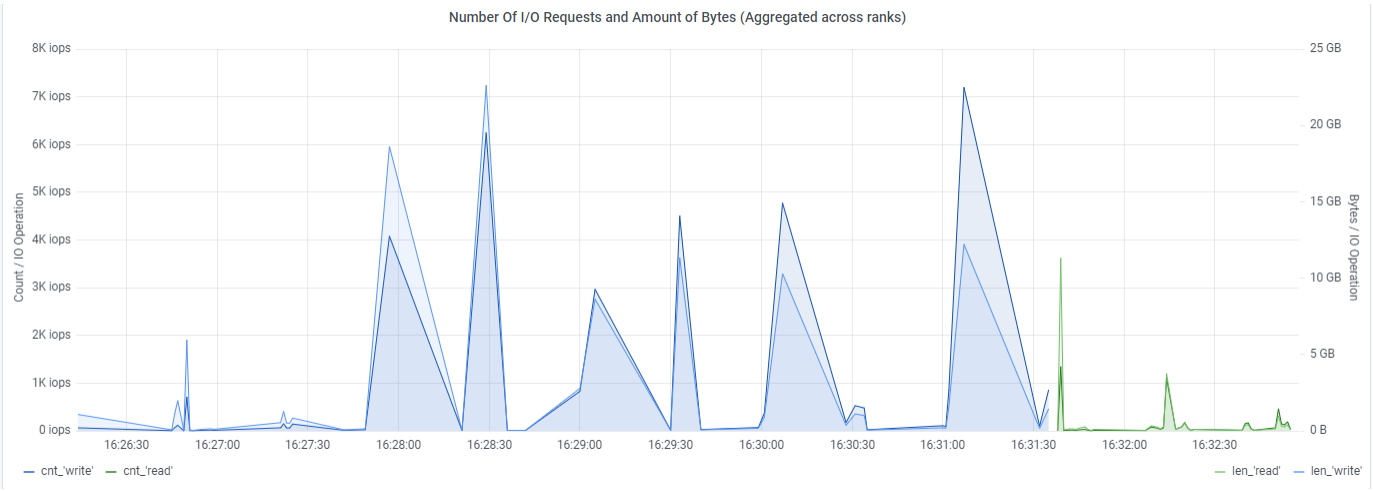

Fig. 9: Graphana visualization of the `job_id 2` writes (blue) and reads (green) operations and amount of bytes per operation, using the absolute timestamp metric collected with *Darshan LDMS Integration*. Given the other metrics collected by the *Darshan-LDMS Connector* , many more analyses and visualizations can be made to allow for further insights in application I/O behavior.

run so as the rate of I/O events increase, the number of json messages formatted will increase which will decrease the applications runtime performance. To address this issue, we will include an option for users to decide the rate of I/O events that the *Darshan-LDMS Connector* will collect and format into a json message to be sent to the LDMS Streams interface. Having this option will allow users who are running intensive I/O applications to still be able to analysis runtime timeseries data of their application without concern of the runtime performance.

We hope the *Darshan LDMS Integration* will be available as an optional ”module” plugin in Darshan so their users may use this tool to better understand their applications I/O performance across HPC systems and clusters.

REFERENCES

[1] E. Costa, T. Patel, B. Schwaller, J. M. Brandt, and D. Tiwari, “Systematically inferring i/o performance variability by examining repetitive job behavior,” in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC ’21. New York, NY, USA: Association for Computing Machinery, 2021. [Online]. Available: https://doi.org/10.1145/3458817.3476186

[2] P. Carns, R. Latham, R. Ross, K. Iskra, S. Lang, and K. Riley, “24/7 characterization of petascale I/O workloads,” in *Proc. 2009 Workshop on Interfaces and Architectures for Scientific Data Storage*, 2009.

[3] “Ovis,” http://ovis.ca.sandia.gov.

[4] “Ovis/SOS,” http://github.com/ovis-hpc/sos.

[5] B. Schwaller, N. Tucker, T. Tucker, B. Allan, and J. Brandt, “HPC system data pipeline to enable meaningful insights through analysis-

[9] ——, “Darshan-runtime installation and usage.” [Online]. Available: https://www.mcs.anl.gov/research/projects/darshan/docs/darshan-runtime.html

driven visualizations,” in *2020 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 2020, pp. 433–441.

[6] G. Labs, “Grafana: The Open Observability Platform.” [Online]. Available: https://grafana.com

[7] “Ovis-wiki,” https://github.com/ovis-hpc/ovis-wiki/wiki, March 2021.

[8] Darshan, “DARSHAN: HPC I/O Characterization Tool.” [Online]. Available: https://www.mcs.anl.gov/research/projects/darshan/

[10] “Ovis/LDMS,” http://github.com/ovis-hpc/ovis.

[11] Darshan, “DARSHAN: HPC I/O Characterization Tool.” [Online]. Available: https://www.mcs.anl.gov/research/projects/darshan/documentation/

[12] “darshan-hpc/darshan,” https://github.com/darshan-hpc/darshan.

[13] S. Habib, V. Morozov, N. Frontiere, H. Finkel, A. Pope, and K. Heitmann, “Hacc: Extreme scaling and performance across diverse architectures,” in *SC’13: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE, 2013, pp. 1–10.

[14] S. Eddy, “Hmmer user’s guide,” *Department of Genetics, Washington University School of Medicine*, vol. 2, no. 1, p. 13, 1992.

[15] E. L. Sonnhammer, S. R. Eddy, E. Birney, A. Bateman, and R. Durbin, “Pfam: multiple sequence alignments and hmm-profiles of protein domains,” *Nucleic acids research*, vol. 26, no. 1, pp. 320–322, 1998.

[16] N. A. Petersson and B. Sjögreen, “Sw4, version 2.0. computational infrastructure for geodynamics,. doi: 10.5281/zenodo. 1045297.”

[17] R. Thakur, R. Bordawekar, A. Choudhary, R. Ponnusamy, and T. Singh, “Passion runtime library for parallel i/o,” in *Proceedings Scalable Parallel Libraries Conference*, 1994, pp. 119–128.

[18] S. J. Kim, S. W. Son, W.-k. Liao, M. Kandemir, R. Thakur, and A. Choudhary, “Iopin: Runtime profiling of parallel i/o in hpc systems,” in *2012 SC Companion: High Performance Computing, Networking Storage and Analysis*, 2012, pp. 18–23.

[19] J. No, S.-S. Park, J. Carretero, A. Choudhary, and P. Chen, “Design and implementation of a parallel i/o runtime system for irregular applications,” in *Proceedings of the First Merged International Parallel Processing Symposium and Symposium on Parallel and Distributed Processing*, 1998, pp. 280–284.