LIOProf: Exposing Lustre File System Behavior for I/O Middleware

Cong Xu *, Suren Byna†, Vishwanath Venkatesan*, Robert Sisneros‡,
Omkar Kulkarni*, Mohamad Chaarawi§, and Kalyana Chadalavada*

*Intel Corporation Email: {cong.xu,vishwanath.venkatesan,omkar.kulkarni,kalyana.chadalavada}@intel.com
†Lawrence Berkeley National Laboratory Email: sbyna@lbl.gov
‡National Center for Supercomputing Applications Email: sisneros@illinois.edu
§The HDF Group Email: chaarawi@hdfgroup.org

***Abstract*—As parallel I/O subsystem in large-scale supercomputers is becoming complex due to multiple levels of software libraries, hardware layers, and various I/O patterns, detecting performance bottlenecks is a critical requirement. While there exist a few tools to characterize application I/O, robust analysis of file system behavior and associating file-system feedback with application I/O patterns are largely missing. Toward filling this void, we introduce Lustre IO Profiler, called LIOProf, for monitoring the I/O behavior and for characterizing the I/O activity statistics in the Lustre file system. In this paper, we use LIOProf for uncovering pitfalls of both MPI-IO's collective read operation over Lustre file system and identifying HDF5 overhead. Based on LIOProf characterization, we have implemented a Lustre-specific MPI-IO collective read algorithm, enabled HDF5 collective metadata operations and applied HDF5 datasets optimization. Our evaluation results on two Cray systems (Cori at NERSC and Blue Waters at NCSA) demonstrate the efficiency of our optimization efforts.**

I. Introduction

Rapidly increasing demands from scientific applications for computational power keeps pushing the High-Performance Computing (HPC) infrastructure towards exascale. While the computing power of HPC systems keeps climbing, the I/O systems have not been able to catch up with the pace of advancement. Worse yet, it is often found that existing applications can only achieve a fraction of the potential I/O performance on large scale platforms [15]. The complexity of I/O systems poses significant challenges in investigating the root cause of performance loss. Potential factors that may prevent the I/O systems from achieving the peak bandwidth include poor I/O access patterns, load imbalance either among the MPI processes or in the underlying file system or both, lock contention, etc.

To cope with such challenges, profiling tools are designed to facilitate the process of I/O characterization in I/O software stacks, generating invaluable statistics for I/O activity analysis and bottleneck detection. I/O operations issued by applications traverse through multiple I/O software layers, including high-level I/O libraries such as HDF5 [12] and Parallel NetCDF [5], MPI-IO middleware, and parallel file system such as Lustre [1]. Multiple application-level I/O instrument tools, such as Darshan [3] and mpiP [14], have been developed to record the I/O operations occurred in high-level I/O libraries and MPI-IO middleware layers, while Lustre Monitoring Tool (LMT) [6] is designed to monitor Lustre server system status in real time.

However, legacy Lustre profiling tools do not provide detailed I/O tracing information for users to conduct systematic analysis. For instance, LMT monitors storage server status, such as CPU utilization, memory usage, and disk I/O bandwidth, but does not provide file system statistics [13]. Among these file system statistics, server-side client statistics, which measure the I/O traffic from each client to the server, provide crucial information for I/O behavior analysis. With this information users are able to get a clear view of I/O requests issued by a specific client within a period of time. Consequently, the workloads on both Lustre servers and clients can be monitored and any load imbalance issues can be easily detected in the system. In addition, so far, the information provided by existing profiling tools is not sufficient to uncover the internal causes of performance degradation, such as disk seeks due to serving multiple client requests concurrently, lock contentions that result from accessing the same portion of the file, and so on.

To address this issue, we propose a tool, called **LIOProf**, for tracking the I/O activities carried out on Lustre file system server, including server-side client I/O statistics, server disk bandwidth being shared between distinct clients, number of lock contentions incurred by a specific client, I/O workload distribution, etc. The goal of LIOProf is to provide as much detailed file system I/O activity as possible to users, helping them investigate the internal causes of any performance degradation on Lustre servers. LIOProf is built based on Lustre RPC Tracing, Lustre I/O kits, as well as system profiling tools.

Using LIOProf, we have evaluated the I/O behavior of Lustre server under various I/O access patterns. The synthetic workload IOR benchmark [8] has been used to perform concurrent collective I/O accesses to a shared file, while LIOProf generates the file system statistics. We compiled IOR benchmark with MVAPICH [7] and Parallel

HDF5 library, and conducted two case studies using IOR MPI-IO and HDF5 APIs, respectively.

In the first case study, we have analyzed the performance of MVAPICH collective buffering read algorithm over Lustre file system. Our evaluation shows that the ROMIO collective write algorithm [10] performs well and is capable of delivering the peak bandwidth. On the contrary, the ROMIO [11] collective read implementation can only achieve a fraction of the Lustre peak bandwidth. Using the I/O statistics generated by LIOProf, we have observed that each OST is serving read requests issued by multiple Lustre clients simultaneously, leading to large number of disk seeks. To overcome this performance degradation, we have implemented a Lustre-aware collective read algorithm, and prototyped this mechanism in the ROMIO implementation of MVAPICH. This implementation was done as part of the Lustre ADIO [9] component in ROMIO. On Cori, a Cray XC40 system located at National Energy Research Scientific Computing Center (NERSC), our implementation outperforms existing MVAPICH by up to 134% using 4096 processes.

In the second case study, we have analyzed the overhead introduced by HDF5 on top of MPI-IO. Parallel HDF5 uses MPI-IO collective buffer algorithm to perform parallel I/O. Thus to quantify the overhead introduced by HDF5, we have compared the performance of HDF5 and MPI-IO APIs using the IOR benchmark. Our evaluation showed that there was a considerable performance gap between parallel HDF5 and MPI-IO cases. We have used LIOProf to identify that metadata write operations issued by IOR with HDF5 was the cause for degraded performance. We have addressed the performance gap by enabling the latest implementation of collective metadata optimizations in HDF5 and by applying dataset creation optimization in the IOR benchmark. These optimizations resolved the inefficiencies of the I/O access pattern of IOR with HDF5.

In this paper, we will present the design and implementation details of LIOProf and demonstrate the use of LIOProf in developing Lustre-aware collective buffer read algorithm and optimizing parallel HDF5 library. To demonstrate the effectiveness of our efforts, we have evaluated our proposed optimizations on the Cori system at NERSC and Blue Waters at NCSA.

II. Background and Motivation

Parallel I/O for scientific computing has gained a lot of traction over the last decade and I/O libraries like MPI-IO [11], HDF5 [12] act as the perfect liaison between a file system and an application. Optimizing such I/O libraries has largely been done using empirical methods. Although empirical approaches can provide insights into the behavior of a library for an application in a specific system configuration, developing and optimizing a generic solution with this approach is challenging. For example, although MPI collective I/O algorithms in theory reduce contention and improve I/O performance by trading-off with communication performance, few get used in leadership scale systems. Several applications often resort custom I/O implementations using file-per-process mode. To this end, PLFS [2] a parallel log structure file system was created to provide an additional level indirection by representing a shared file with a set of non-shared files in an underlying file system like Lustre. Although file-per-process approach can work currently, it will be challenging to use the same approach on exascale systems where metadata complexity can grow to be significantly higher. It is important to improve shared file I/O performance to a point where it could be used at all times by default. One reason for poor performance of collective I/O algorithms is that they treat the underlying file system as a black-box. Feedback from the file system on the behavior of a collective I/O algorithm can help understanding and optimizing the performance. LIOProf is primarily focused on achieving this.

All client server communications in Lustre [1] are coded as RPC (remote procedure call) requests and return. All data requests are handled by Lustre Object Storage Servers (OSSes) where each server can have one or more storage targets (OSTs). Performance of a shared file depends on the distributing contiguous blocks of a file, called a stripe, across Lustre OSTs. A file properly striped across multiple OSTs is able to obtain closer to the achievable peak bandwidth of underlying file system in contrast to a scenario where there is contention at OSTs. The number of stripes a file is striped on is called *stripe count*, and the size of a block is called *stripe size*. Contention may occur when the stripe sizes are chosen inappropriately leading to stripes from all writer processes ending up in more than one OST, thereby constricting the parallelism.

An MPI collective I/O algorithm that does not take the striping information of a file into account would result in poor performance, despite using aggregation at the clients. Unfortunately, such level of understanding is hard to achieve from empirical data. With LIOProf, our goal is to capture RPCs sent for data requests in Lustre and to visualize the imbalance on OSTs, which is key to identify the right stripe size to be used in collective algorithms. LIOProf can also be used with other I/O libraries like HDF5. For example, it can also help identifying the amount of padding required for custom metadata to be stored along with files for scientific libraries like HDF5. By providing useful insights into the behavior of Lustre file system, LIOProf will be helpful in bridging the gap between parallel I/O libraries, middleware, and parallel file systems.

III. Related Work

There have been several efforts to study I/O access patterns in parallel applications on Lustre. These range from using profiling libraries for instrumenting binaries, to monitoring the file system itself. MpiP [14] is a light-weight

profiling library for MPI applications that intercepts MPI calls using the profiling API and generates statistics such as the call count and the time spent in calls. Profiling is useful for locating bottlenecks by identifying functions that take longer to execute and make up a significant portion of the overall execution time. However, it provides no information about how the application interacts with the file system. The I/O characterization tool Darshan [3], on the other hand, is an excellent tool for studying access patterns. It provides great insights into I/O activity on the client side, which includes plots of transfer sizes, temporal data distribution, data rates, IOPS frequencies, and many more.

While client-side profiling reveals I/O behavior from an application standpoint, monitoring the file servers is much more helpful in understanding how applications interact with the file system. The Lustre Monitoring Tool (LMT) [13] is very handy for observing Lustre usage statistics in real time. It is configured as a Cerebro plugin running on Lustre servers that collects aggregate statistics and stores it in a MySQL database, which can then be queried for displaying the usage statistics to the end user. LMT comes packaged with two command-line tools *lstat* and *ltop* as well as a GUI-based tool *lwatch* for visualization of statistics, all coded in Java. While LMT collects some useful samples from the OSS server nodes, it focuses extensively on collecting statistics from the Lustre Metadata Servers (MDS). As the description of LMT [6] notes, the results collected by the LMT may be affected by the transfer size, Lustre RPC size and use of collective I/O wherein only a subset of the clients interact with Lustre. The Robinhood Policy Engine [4] for Lustre, in addition to triggering actions on the file system in response to events, also generates highly detailed reports and statistics.

Client-side profiling and server-side monitoring are useful to study the corresponding I/O patterns within an application and a file system, respectively, but are not suitable for observing how application I/O requests correlate with file system activities. This motivates us to create LIOProf to correlate application I/O requests with the server-side activity.

IV. LIOProf

LIOProf is designed to trace I/O activities taken place in Lustre file system. It aims to provide detailed I/O tracing information and status of Lustre OSS servers. We have implemented LIOProf using multiple sets of scripts without requiring any modification to the Lustre kernel. These scripts invoke Lustre RPC Tracing and system plug-in services, and generate information that can be analyzed to shed light on the I/O behavior being carried out in the file system.

In order to mitigate the overhead introduced to the Lustre file system, LIOProf uses post analysis strategy to eliminate any performance impact. During the run time of the job execution, statistical metrics are tentatively stored on the local storage of the Lustre OSS servers. The advantage of this approach is that the traces of all the OSS servers do not occupy network for global information exchange and synchronization. To coordinate the data source for characterizations, all the trace messages are marked with the simultaneous time on the server. This can be leveraged for the aggregation of the statistics.

LIOProf contains two main components: *LIOProf Logging Services* and *LIOProf Statistics Collection and Visualization*. As shown in Fig. 1, LIOProf Logging Services are launched on Lustre OSS server nodes before the execution of applications. These services are mainly responsible for recording the I/O activities and the status of each OSS server node. Once the application finishes its job, LIOProf Statistics Collection and Visualization component collects the statistical metrics from LIOProf Logging output, aggregates the gathered data, and generates visualization plots for manual analysis.

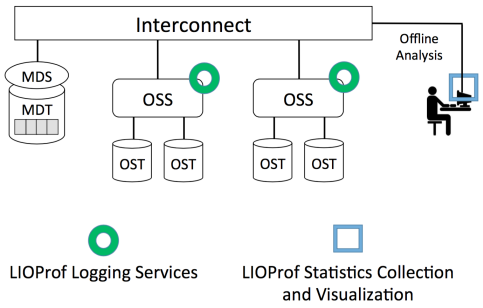

Figure 1. A high-level overview of LIOProf Components

A. LIOProf Logging Services

To gain insights into the I/O characteristics of the Lustre OSS nodes, LIOProf enables Lustre RPC Tracing and spawns plug-in status services on user specified OSS nodes. Before the execution of the application, the LIOProf Logging Services need to be started manually on Lustre by someone with administrator access. At the moment, we assume that LIOProf is to be used only by facility administrators to identify bottlenecks of I/O access patterns on Lustre, which in turn assumes that Lustre is exclusively used by a single application.

Lustre RPC Tracing can be enabled by setting parameter debug to be rpctrace log level. The historical RPC Tracing logs need to be cleared before the execution of an application. A debug log buffer is used to store RPC Tracing logs temporarily. We increase the log buffer size to prevent the loss of the logs due to overflow. A background debug_daemon is employed to drain the logs from the debug log buffer on the fly. This daemon is recommended to be used when the logs are needed for an extended period of time. In addition, users of LIOProf can also get log

[Lustre RPC Log **Format**]

message_mask:subsystem_mask:cpu_number:time:0:servicing thread PID:0:(sourcefile:line number:function name)
MESSAGE

[Lustre RPC Log **Example**]

00000100:00100000:27.0:1458939850.660586:0:16998:0:(service.c:2071:ptlrpc_server_handle_request())
Handling RPC pname:cluuid+ref:pid:xid:nid:opc
ll_ost_io05_000:f2c9a9e1-3fbb-9182-59db-ca8c14b411d7+7:8424:x1529178798258480:12345-192.168.1.38@o2ib:3

Figure 2. Lustre RPC LOG

complaints if the logs are dropped. Once the application finishes its execution, the logs in the buffer are forced to be flushed out and the debug_daemon is terminated.

Besides tracing Lustre RPC logs, LIOProf is able to measure the maximum available bandwidth of the Lustre file system and to characterize the I/O rates of the hard drives. Obdfilter-survey benchmark is usually used to investigate the obdfilter layer in Lustre IO stack for writing, rewriting and reading multiple Lustre objects. To obtain the peak performance of Lustre OSTs, LIOProf can by leveraged by users to launch Obdfilter-survey and output the optimal performance of Lustre system. $Iostat$ and brw_stats are two crucial tools used by LIOProf to monitor and understand the status of storage devices. $Iostat$ is part of the $sysstat$ family of tools that takes a snapshot of the status of specified device. LIOProf lets $iostat$ capture the disk bandwidth in a one second interval and outputs the statistics to local storage. brw_stats is one of the statistics provided in the $/prof/fs/lustre/*$, that indicates the number of contiguous I/O accesses cumulated during a period of time. LIOProf leverages brw_stats to summarize the Lustre RPCs information. Each service will create one independent file on each OSS node for recording statistics.

Since LIOProf only launches a small number of logging and status services, this will not introduce too much overhead to the system. To quantify the overhead introduced by LIOProf, we have conducted multiple experiments under various I/O access patterns. During the run time of the job, LIOProf added less than 1% overhead for the I/O operations.

B. LIOProf Statistics Collection and Visualization

LIOProf Statistics Collection and Visualization component is designed to visualize I/O statistics of the traces LIOProf Logging Services collected from the Lustre OSS nodes. One can run the parsing and visualization scripts anywhere in an offline manner as shown in Fig. 1. LIOProf uses high-performance, parallel remote shell tool *pdsh* to facilitate the collection of the I/O logs across all the OSS nodes after the execution of the application. These Lustre RPC Tracing logs and server status are parsed and the generated outputs are collected and organized for *gnuplot* to draw visualization and analysis of the characteristics.

Lustre RPC Tracing logs record thousands of events occurred on Lustre OSS nodes, LIOProf only extracts some of useful information from these logs for statistical analysis and visualization. Among these logs, RPC messages are the most important information provided by Lustre kernel. Fig. 2 shows the Lustre RPC log format and an example log message. RPC log consists of message mask, subsystem mask, cpu number, message time, message content, etc. For instance, from the example in Fig. 2, we can see the RPC log was handed by cpu 27 at system time 1458939850.660586, and the log was dumped by a $ptlrpc\_server\_handle\_request()$ function. LIOProf uses the log time to accumulate and report the number of RPC requests issued within each time interval. In addition, this facilitated time information can be used to synchronize the logs among various OSS nodes.

The MESSAGE content contains the details of the message, including source of the RPC, RPC operation code (RPC opc), and so on. The source of the RPC provides useful information for tracking the RPC requests issued by specific Lustre Client. With this information, LIOProf can easily generate Lustre server side clients' requests for users. Lustre RPC operation code (RPC opc) indicates the type of RPC message, it is defined in header file "lustre_idl.h". LIOProf leverages RPC opc to identify the requests from the clients, such as I/O requests and lock requests.

Once the logs have been collected and parsed, the intermediate results will be passed to the gnuplot script to draw the stuff. LIOProf is able to generate multiple types of figures, including I/O distributions across Lustre server, server-side client I/O statistics, number of lock contentions triggered in the system and so on.

V. RESULTS

We have deployed LIOProf on a development cluster, called Wolf, at Intel. The Wolf cluster contains 70 nodes, each equipped with Octadeca-Core 2.3GHz Xeon processors (36 Cores), 64 GB memory, and six 1TB SATA disks. These nodes are connected using Mellanox QDR ConnectX InfiniBand. Analysis of even these early I/O characterizations revealed a seemingly addressable performance issue with the standard MPI-IO implementation on Lustre. In

this section we will describe the scenario that directed our I/O optimizations efforts and demonstrate the real and significant performance improvements we were able to gain on capability systems in current production, such as Cori at NERSC and the Blue Waters system at the National Center for Supercomputing Applications (NCSA). We extended subsequent testing to the high-level I/O library, HDF5, to ensure our low-level library improvements would be widely applicable. Below we will also describe associated additional overheads and how we addressed these.

The Cori system at the National Energy Research Scientific Computing Center (NERSC) has 1630 compute nodes and 30PB of Lustre storage. One node provides 32 CPU cores and 128GB of memory. Cori uses the Cray Aries high-speed interconnect with a Dragonfly topology. Blue Waters is a Cray XE6-XK7 supercomputing system managed by NCSA. The system has 26 PB online disk capacity and two types of compute nodes: XE6 and XK7. There are 22,640 XE6 nodes and 4,224 XK7 nodes connected via Cray Gemini interconnect. Our tests were run on the XE6 portion of the machine; those nodes have two 16 core CPUs and 64GB of main memory.

We have used the IOR benchmark for our evaluations in this paper. IOR is a flexible tool for emulating diverse I/O access patterns using different I/O libraries, including POSIX, MPIIO, and HDF5. IOR is widely used for investigating I/O library configurations on Lustre file system. We have used the MVAPICH2 framework, version 2.2b and tested on Lustre versions 2.7 and 2.5.1, the version on Blue Waters. We have applied our MPI-IO optimizations using the same version of MVAPICH2 framework.

A. An Improved Collective Read Algorithm for Lustre

To investigate the MPI-IO performance over Lustre file system, 192 processes are employed to perform concurrent I/O in an interleaved access pattern on a shared file. Regarding the IOR configuration, the total size of the data is 768GB, which is twice the size of memory on Lustre clients. To avoid lock contentions, both IOR block size and transfer size are set to be 4MB, which equals the stripe size of the Lustre. For the Lustre configuration, 6 Lustre clients access 4 Lustre OSTs remotely through high speed interconnect. The stripe count is configured to be 4, thus the measured file is distributed to all the Lustre OSTs.

In the rest of the paper, Obdfilter-survey is employed by LIOProf to obtain the maximum available bandwidth of the Lustre file system. The aim of our optimization efforts is to close the performance gap between full I/O benchmarks and Obdfilter-survey results.

Benchmarking core ROMIO algorithms of the MPI-IO implementation on the Wolf cluster shows a significant disparity between collective write performance with that of collective read. In Fig. 3, we show that the MPI-IO collective write of IOR performs close to the maximum available bandwidth. On the contrary, the collective read performance achieves only about half of the Obdfilter-survey bandwidth. Understanding the relatively under-performing read was our first use case for the collection and analysis of LIOProf I/O statistics.

In Fig. 4, we show the I/O requests traced by LIOProf for the read benchmark, where we observed poor performance. From this figure, we can see that each MPI process (client) is accessing all the OSTs and the bandwidth of the OSTs is split across all the processes. Since each process reads from a different region of the data in this access pattern, each OST has to read data from different regions on the disks. In other words, an OST has to seek data from different locations for serving all the MPI processes. Disk seeks are notoriously costly and result in poor overall bandwidth from each OST. An optimization would be to limit the number of processes accessing an OST and reduce the number of disk seeks.

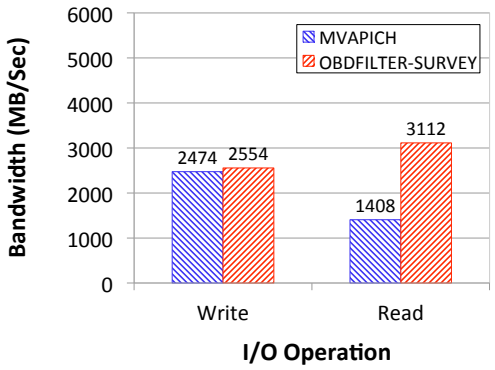

Figure 3. Simple benchmark of MVAPICH collectives highlighting the performance gap between reads and writes.

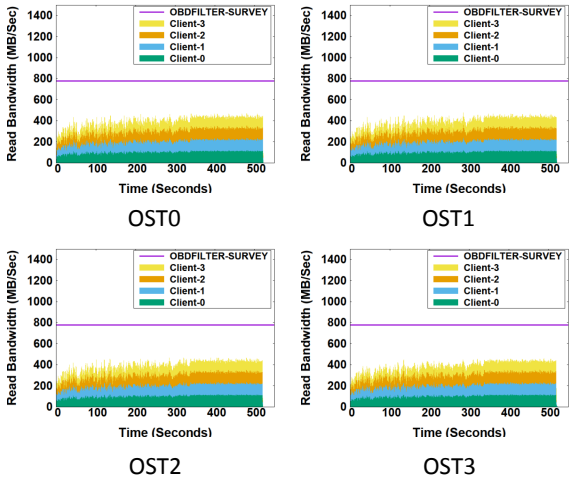

Figure 4. The distribution across all Lustre clients for each OST. This data is available from an I/O characterization generated by LIOProf.

We have implemented a Lustre-aware collective read algorithm where Lustre striping information is used to ensure one process is dedicated to reading data from one OST. This process then forwards the data to other processes when necessary. We have prototyped this mechanism in the ROMIO implementation of MVAPICH2. Fig. 5 shows an I/O characterization of this implementation. As a result, OST contentions are mitigated leading to fewer disk seeks and therefore reduced overhead. Fig. 6 demonstrates the resulting performance increase gained by our Lustre-aware read. As shown in the figure, the optimized implementation is able to deliver 2874 MB/Sec bandwidth; a performance now akin to the collective write algorithm, near the maximum bandwidth of the Lustre file system.

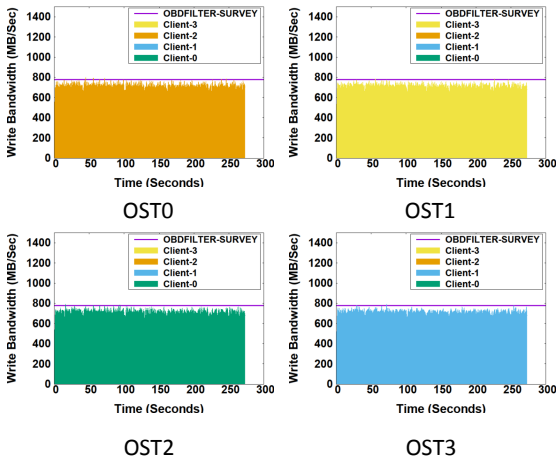

Figure 5. I/O characterization of the enhanced MVAPICH2 (with Lustre-aware collective read) generated by LIOProf.

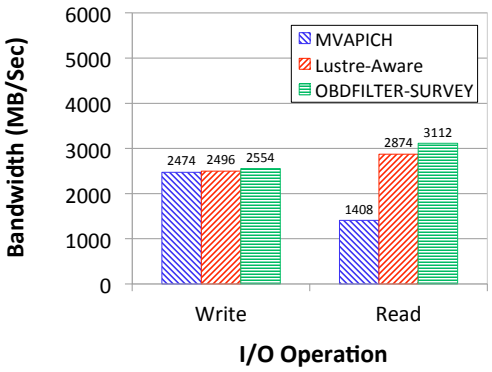

Figure 6. Performance improvement resulting from our Lustre-aware collective read algorithm.

We have tested the new Lustre-aware collective buffering read on production systems to verify the performance. On the Cori supercomputer, we have tested IOR with different numbers of processes from 128 to 4096 using 128 Lustre Clients. These processes perform a collective read of 16TB of data from 96 Lustre OSTs. From Fig. 7 we can see, the Lustre-aware collective buffering read algorithm performs 1.34X faster than the original implementation at 4096 processes. As shown in Fig. 8, the highest scale test was replicated on Blue Waters where we saw a 2.70X speedup in Lustre collective read performance. In addition to the persistence of reduced disk seeks we observed on our test system we believe we are also benefiting from higher read cache utilization. The performance of our Lustre-aware read algorithm is similar to Cray's optimized version (see Fig. 7), but we are unaware of implementation similarities as that algorithm is closed source.

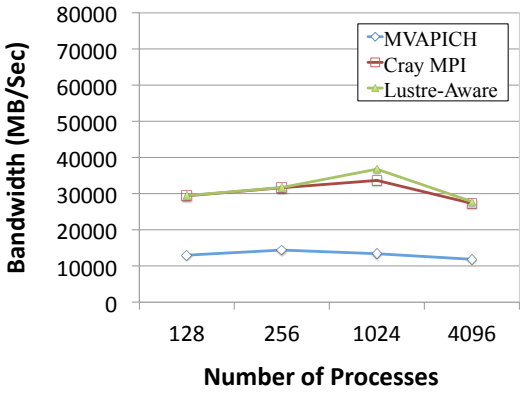

Figure 7. Performance characteristics of the original, Cray, and our Lustre-aware read algorithms up to 4096 cores on Cori.

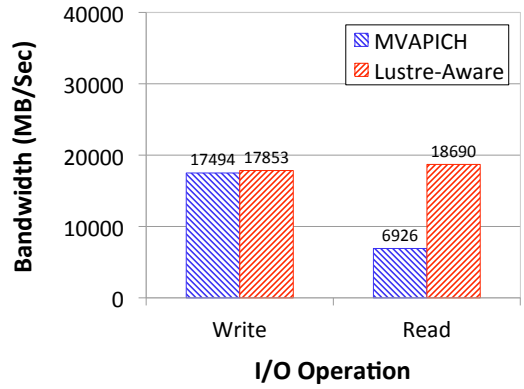

Figure 8. Performance characteristics of the original and our Lustre-aware read algorithms up to 4096 cores on Blue Waters.

B. Improving performance of IOR with HDF5

After our optimization of the collective read algorithm, MPI-IO layer is able to deliver near optimal performance. We then investigated the efficiency of IOR in using HDF5 with our LIOProf tool. Parallel HDF5 depends on MPI-IO

Figure 9. Performance of IOR with MPI-IO and HDF5

to perform collective I/O operations. We have compared the performance of IOR using MPI-IO and HDF5 on the Wolf cluster. To simplify our analysis, we create the ideal setup in our evaluation. 4 processes are launched on 4 Lustre clients, perform concurrent I/O to a shared file distributed across 4 Lustre OSTs. The total size of the file is 512GB, and IOR block size, transfer size and Lustre stripe size are configured to be 4MB.

In Fig. 9, we compare the performance of IOR with MPI-IO API and that with HDF5 API. As shown in the figure, the IOR benchmark using MPI-IO significantly outperforms HDF5, nearly doubling HDF5 for collective reads.

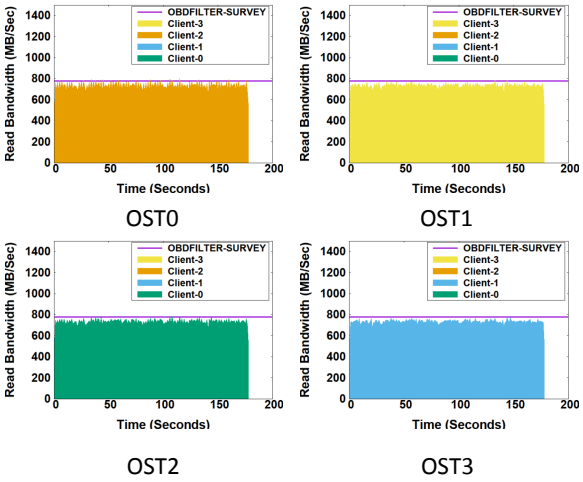

Figure 10. MPI-IO I/O characterization

To find out the cause of the performance difference, we have used LIOProf to characterize the I/O activities on Lustre OSS nodes. Fig. 10 shows the number of RPC Read Requests issued by Lustre Clients on each OST in the MPI-IO case. As shown in the figure, each OST serves one Lustre Client and is able to deliver high I/O bandwidth consistently. The default Lustre RPC size of 1MB is used in this evaluation and each OST is able to deliver 778MB/s bandwidth on average.

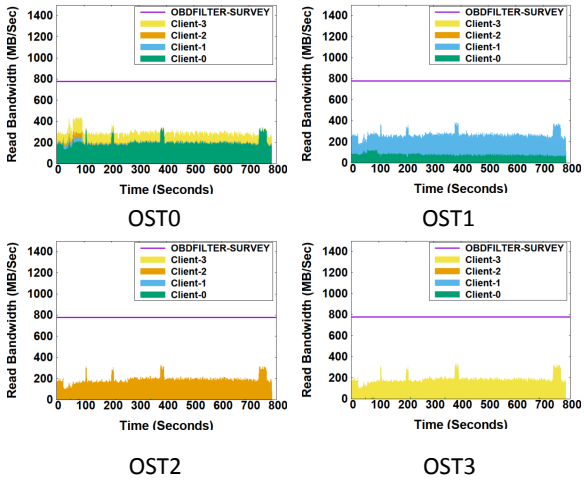

Figure 11. Original HDF5 I/O characterization

In Fig. 11, we show the number of I/O requests issued on Lustre OSSs in HDF5 case. Both OST0 and OST1 handle I/O requests from multiple Lustre Clients, and the aggregate bandwidth is about 300MB/s. In the IOR benchmark, parallel HDF5 uses the MPI-IO collective buffering algorithm while it adopts MPI-IO independent I/O method for metadata operations. The blue filled curves in OST1 subfigure shows the I/O requests for reading HDF5 datasets and the green filled curves represents the HDF5 metadata requests. The statistics provided by LIOProf allow us to pinpoint HDF5 metadata I/O accesses as the primary factor in the observed reduction in I/O bandwidth.

The HDF Group has recently implemented a (currently unreleased) collective metadata I/O feature. This implementation uses MPI collective operations to perform collective metadata I/O accesses. On metadata reads, one process is selected to read metadata and that process broadcasts it to all other processes, leading to fewer small I/O accesses. Collective metadata write efficiency is also improved by calling collective write function for each dataset. We have modified the IOR benchmark to test these prerelease collective metadata operations, H5Pset_coll_metadata_write() and H5Pset_all_coll_metadata_ops().

Fig. 12 shows the performance of HDF5 with collective metadata optimization. As observed in the figure, both write and read performance of HDF5-Coll_Meta cases have been improved when compared with the original HDF5 cases. In the collective metadata read operation only one process is selected to read metadata with other processes receiving their metadata from the chosen process without touching the storage. LIOProf indeed reveals, as shown in Fig. 13, the number of I/O requests have been significantly cut down.

However, there is still a noticeable performance gap

Figure 12. Overall Performance of HDF5 with collective metadata

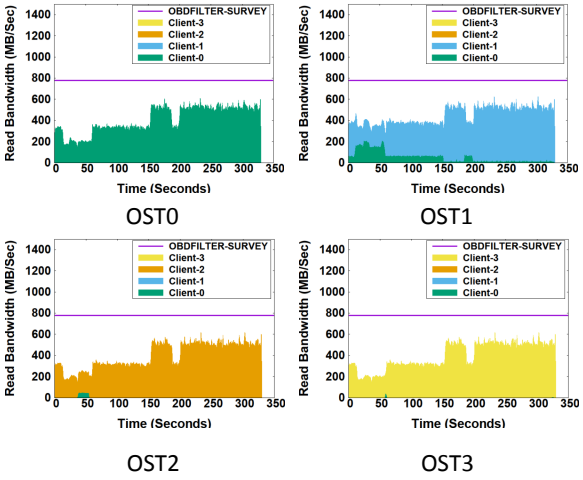

Figure 13. HDF5 with collective metadata I/O characterization

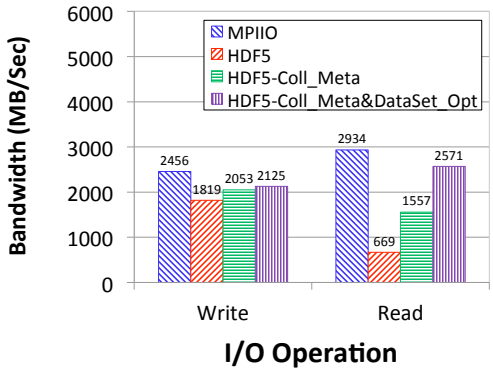

Figure 14. Overall Performance of HDF5 with collective metadata and datasets optimization

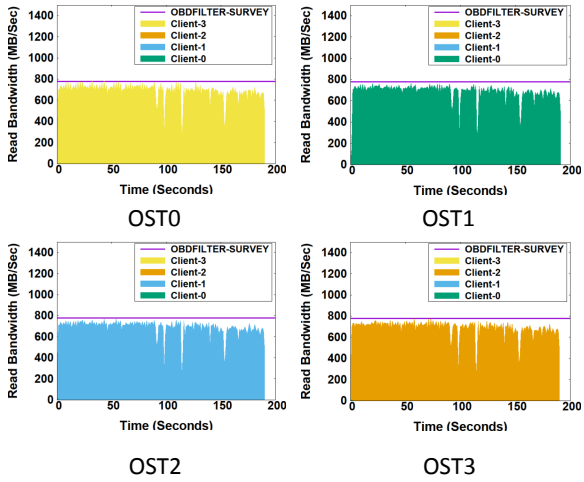

Figure 15. HDF5 with collective metadata and datasets optimization I/O characterization

between MPI-IO case and HDF5-Coll_Meta case. Fig. 13 reveals the collective metadata I/O accesses still hinder HDF5 from achieving performance levels of the MPI-IO API. Upon examination of the IOR benchmark code with HDF5 we found that IOR creates thousands of datasets for the performance measurement, and the program needs to access HDF5 metadata for each dataset before accessing the actual data. To address this issue, we modified the IOR source code to obtain and then cache metadata information in the beginning of the program to mitigate the negative impact of metadata operations on the I/O bandwidth.

After enabling collective metadata and applying dataset metadata caching optimizations, we further increase the I/O bandwidth of the IOR benchmark with HDF5, especially for the read operation. As shown in Fig. 14, the read bandwidth of HDF5-Coll_Meta-DataSet_Opt case performs 65.1% and 284.3% better than HDF5-Coll_Meta and original HDF5 cases, respectively. Fig. 15 characterizes the I/O activities generated by the LIOProf monitoring tool. In that figure we can see that without the interference of metadata operations our optimizations allow each OST to deliver high bandwidth consistently.

VI. Conclusion

Despite several advances in parallel file systems and parallel I/O software layers, it is still complex to obtain peak capacity on the parallel file systems. Client-side profiling tools, such as Darshan, and file system monitoring tools, such as Lustre, provide only a partial picture of the I/O performance. To fill the void of correlating an application's I/O access pattern and its performance with file system behavior, we introduce LIOProf in this paper. Using LIOProf, we observed poor performance of MPI-IO two-phase (collective buffering) read algorithm that resulted in distributed I/O requests to all the participating Lustre OSTs. We have implemented a Lustre-aware MPI-IO collective read operation that reduced the number of disk seeks on each OST and

hence improved I/O performance. We have also studied the performance of IOR benchmark with HDF5, where the original implementation of IOR has several inefficiencies in accessing metadata. By using a new collective metadata optimization in HDF5 and by caching the HDF5 dataset operations in IOR, we have observed HDF5 performing closer to our improved MPI-IO.

VII. Acknowledgments

We are very thankful for the contributions of Eric Barton and Oleg Drokin from Intel. This work is supported by the Director, Office of Science, Office of Advanced Scientific Computing Research, of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231. This research used resources of the National Energy Research Scientific Computing Center. This is also funded in part by the Blue Waters sustained-petascale computing project, which is supported by the National Science Foundation (awards OCI-0725070 and ACI-1238993) and the state of Illinois.

References

[1] Lustre 2.0 operations manual. http://wiki.lustre.org/images/3/35/821-2076-10.pdf.

[2] J. Bent, G. Gibson, G. Grider, B. McClelland, P. Nowoczynski, J. Nunez, M. Polte, and M. Wingate. Plfs: A checkpoint filesystem for parallel applications. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, pages 21:1–21:12, New York, NY, USA, 2009. ACM.

[3] P. H. Carns, R. Latham, R. B. Ross, K. Iskra, S. Lang, and K. Riley. 24/7 characterization of petascale i/o workloads. In *CLUSTER*, 2009.

[4] T. Leibovici. Taking back control of hpc file systems with robinhood policy engine. *CoRR*, abs/1505.01448, 2015.

[5] J. Li, W.-k. Liao, A. Choudhary, R. Ross, R. Thakur, W. Gropp, R. Latham, A. Siegel, B. Gallagher, and M. Zingale. Parallel netcdf: A high-performance scientific i/o interface. In *Proceedings of the 2003 ACM/IEEE Conference on Supercomputing*, SC '03, pages 39–, New York, NY, USA, 2003. ACM.

[6] C. Morrone. LMT Lustre Monitoring Tools. In *Lustre User Group Conference*, 2011.

[7] D. K. Panda. MVAPICH: MPI over InfiniBand, 10GigE/iWARP and RoCE. 2015.

[8] H. Shan and J. Shalf. Using IOR to analyze the I/O performance for HPC platforms. In Cray Users Group Meeting (CUG) 2007, Seattle, Washington, USA, 2007.

[9] R. Thakur, W. Gropp, and E. Lusk. An abstract-device interface for implementing portable parallel-i/o interfaces. In *IN PROCEEDINGS OF THE 6TH SYMPOSIUM ON THE FRONTIERS OF MASSIVELY PARALLEL COMPUTATION*, pages 180–187. IEEE Computer Society Press, 1996.

[10] R. Thakur, W. Gropp, and E. Lusk. Data sieving and collective i/o in romio. In *Frontiers of Massively Parallel Computation, 1999. Frontiers '99. The Seventh Symposium on the*, pages 182–189, Feb 1999.

[11] R. Thakur, W. Gropp, and E. L. Lusk. On implementing mpi-io portably and with high performance. In *IOPADS*, 1999.

[12] The HDF Group. Hierarchical Data Format Version 5. https://www.hdfgroup.org/HDF5/, 2016.

[13] A. Uselton. Deploying Server-side File System Monitoring at NERSC. In *Cray User Group Conference*, 2009.

[14] J. Vetter and C. Chambreau. mpiP: Lightweight, Scalable MPI Profiling. In *CLUSTER*, 2014.

[15] V. Vishwanath, M. Hereld, V. Morozov, and M. E. Papka. Topology-aware data movement and staging for i/o acceleration on blue gene/p supercomputing systems. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 19:1–19:11, New York, NY, USA, 2011. ACM.