I/O Bottleneck Detection and Tuning: Connecting the Dots using Interactive Log Analysis

Jean Luca Bez*, Houjun Tang*, Bing Xie†, David Williams-Young*,
Rob Latham‡, Rob Ross‡, Sarp Oral†, Suren Byna*
* Lawrence Berkeley National Laboratory, † Oak Ridge National Laboratory, ‡ Argonne National Laboratory

***Abstract*—Using parallel file systems efficiently is a tricky problem due to inter-dependencies among multiple layers of I/O software, including high-level I/O libraries (HDF5, netCDF, etc.), MPI-IO, POSIX, and file systems (GPFS, Lustre, etc.). Profiling tools such as Darshan collect traces to help understand the I/O performance behavior. However, there are significant gaps in analyzing the collected traces and then applying tuning options offered by various layers of I/O software. Seeking to connect the dots between I/O bottleneck detection and tuning, we propose *DXT Explorer*, an interactive log analysis tool. In this paper, we present a case study using our interactive log analysis tool to identify and apply various I/O optimizations. We report an evaluation of performance improvement achieved for four I/O kernels extracted from science applications.**

I. Introduction

The HPC I/O stacks deployed in large-scale computing centers expose a plethora of tunable parameters and optimization techniques that can be accessed by API calls, both seeking to improve performance. However, there is little guidance to developers and end-users on how and when to apply them. There is often a lack of knowledge that those options are available or could help in a particular scenario. Furthermore, there has not been a single set of instructions to compile a set of tuning parameters. Coming up with a list of best practices even for a single system is challenging due to various factors, including the size of I/O requests, communication and synchronization costs at the MPI-IO level, file system, performance variability, and storage characteristics.

The tuning options and advanced APIs provided by the HPC I/O stack are often unexplored. For instance, some options that users often do not tune include the parallel file system (PFS) striping settings, MPI-IO hints, and I/O library optimizations [1]–[4]. If we look at Lustre (PFS), modifying the size in which a file is striped with, or the number of storage servers it is striped over to be the best for a given system is not very often considered [5]. For instance, on the Cori system at the National Energy Research Scientific Computing Center (NERSC), Darshan logs from January 2019 report that $\approx 94\%$ of the files used the default 1 MB stripe size and $\approx 36\%$ are striped over a single storage server. MPI-IO can also take advantage of hints provided by users to enable and tune the I/O performance, such as collective buffering and data sieving. High-level I/O libraries, such as HDF5, also expose APIs to enable specific optimizations that users often fail to explore.

Between a performance bottleneck and its tuning solution, there remain dots to be connected that are not fully addressed by existing research efforts. For instance, collective buffering and data sieving were implemented more than 20 years ago [6], yet, we still find it challenging to choose the number of aggregators, their placement, and matching them according to the concurrency at the file system level. Furthermore, combining optimization in multiple levels and finding the best set of tunable parameters to achieve performance can be cumbersome. That alone is the target of different approaches [2], [3], [7], [8]. The challenge is not only the search space exploration and time required to do it, but also the inter-dependencies between various I/O software layers (i.e., HDF5, MPI-IO, and file systems), and how to detect bottlenecks. However, we found that an interactive exploration of the application's profile and traces using Darshan [9] logs already provides insights to connect all the dots and bring the performance to another level.

In this paper, we propose an interactive log analysis process of identifying I/O performance bottlenecks, mapping them to potential problems, and applying tuning options to obtain good I/O performance using *DXT Explorer* (`github.com/hpc-io/dxt-explorer`). We present a case study with four I/O kernels from HPC applications that exercise various I/O tuning parameters and report the performance. We also discuss the gaps in the process of connecting the dots between bottleneck identification and tuning.

II. Background

In this study, we used HPC I/O stacks that use HDF5 as a high-level I/O library, MPI-IO as the I/O middleware, and two file systems (Lustre and GPFS) on two production supercomputing systems: Cori at NERSC and Summit at OLCF. We use Darshan profiling with extended tracing (DXT) for analyzing I/O performance.

HDF5 (Hierarchical Data Format Version 5) is a well-known self-describing file format and an I/O library [10] that provides flexibility, extendibility, and portability. HDF5 is used widely in many science domains as a *de facto* standard to manage various data models [11]. MPI-IO featured as part of the MPI-2 standard provides a comprehensive API and optimization features such as collective buffering and data sieving to perform efficient parallel I/O [6]. HDF5 uses the MPI-IO layer to perform parallel I/O.

A. Parallel File Systems in Production Supercomputers

1) Lustre on Cori: Cori is a Cray XC40 supercomputer deployed at NERSC. It is comprised of $2,388$ Intel Xeon Haswell processor nodes and $9,688$ Intel Xeon Phi Knight's Landing (KNL) compute nodes, all connected to a Lustre parallel file system capable of storing ≈ 30 PB and achieving a peak I/O bandwidth of 744 GB/s. The parallel file system is exposed as a single POSIX namespace with five Metadata Servers (MDSes) and 244 Object Storage Servers (OSSes). Each MDS is responsible for a portion of the global namespace and each OSS manages one Object Storage Target (OST).

2) GPFS on Summit: Summit is an IBM-built supercomputing system deployed at OLCF, containing 4,608 compute nodes. Summit is connected to Alpine, a 250 PB Spectrum Scale (GPFS) file system with a peak bandwidth of 2.5 TB/s. Alpine is comprised of 154 Network Shared Disk (NSD) servers, each NSD manages one GPFS Native RAID (GNR), and serves as both a storage and metadata server. Alpine is configured with a default block size is 16 MB, and unlike Lustre, users cannot change this parameter.

B. I/O Tracing with Darshan

Darshan [9] is a popular tool to collect I/O profiling information from applications. Darshan aggregates I/O profile information to provide valuable insights without adding overhead or perturbing application behavior. Darshan also provides an extended tracing module (DXT) [12] to obtain a fine-grain view of the application behavior to understand I/O performance issues. Once enabled, DXT collects detailed traces from the POSIX and MPI-IO layers reporting the operation (write/read), the rank that issued the call, the segment, the offset in the file, and the size of each request. It also captures the start and end timestamps of all the operations issued by each rank. In this paper, we harness the DXT module to help us analyze and identify I/O performance issues.

III. Interactive Exploration & Optimization

While I/O profiling tools, such as Darshan [9] and Recorder [13], can provide I/O traces, there is a significant gap between the trace analysis and the tuning process. DXT logs provide detailed information about the MPI-IO and POSIX calls. However, there is a lack of tools to analyze the logs and guide tuning parameters setting. Furthermore, existing tools do not offer a straightforward way to analyze and interactively visualize the I/O behavior reported in the DXT logs. The collected trace can be huge for applications with numerous small I/O requests or a long runtime, making them difficult to explore and visualize. Darshan does not provide a way to visualize DXT traces, and manual static plots to analyze DXT logs are limited in the information they present due to space constraints and pixel resolution, possibly hiding I/O bottlenecks in plain sight. Though we can visualize traces captured from Recorder in web-based plots using the *recorder-viz* Python library[1], this solution does not provide relevant contextual information for the I/O operations in the plots. It also does not allow users to explore the traces in time (and in a coordinated way) when using POSIX and MPI-IO interfaces.

Toward filling this gap and aiding the process of I/O log analysis, we introduce ***DXT Explorer***, an interactive tool with zoom-in/out capabilities. The tool provides a coarse-grain and a fine-grain view of the observed I/O behavior from DXT logs.

[1]https://pypi.org/project/recorder-viz

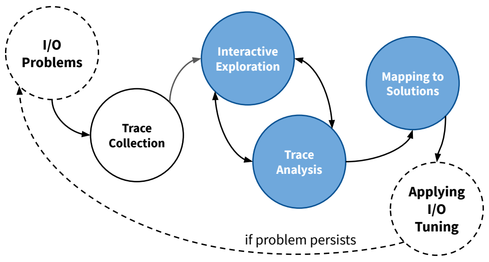

Figure 1: Iterative workflow to identify I/O performance issues based on the interactive visualization of DXT traces.

The global view of DXT Explorer makes it easy to pinpoint issues related to collective data and metadata operations by visual comparison of the behavior at the MPI-IO and POSIX levels. It also offers an analysis process needed in connecting the dots between the I/O problem detection to the tuning strategies (as shown in Figure 1), by using interactive trace exploration and analysis to map problems to existing solutions.

DXT Explorer takes a DXT log (`.darshan` file) as input to parse and transform the traces into interactive web-based visualizations using R with ggplot2 [14] and Plotly [15] libraries. We show a high level output of the tool in Figure 2. The interactive visualization allows a user to ① dynamically narrow down the plot to cover a time interval of interest or ② zoom into a subset of ranks to understand the I/O behavior. We depict two synchronized facets: the first representing the MPI-IO level, and the second, how it was translated into the POSIX level. For each operation, by hovering over the depicted interval, it is possible to inspect additional details ③ such as the type, runtime, MPI rank, and transfer size in KB. The tool also allows to explore the spatiality of accesses in the file and correlate the total transfer size with rank over time. A few example interactive output plots are available at `jeanbez.gitlab.io/pdsw-2021`.

By visualizing the application behavior, we are one step closer to understand the existence of any performance bottlenecks and to apply the most suitable set of optimization techniques to improve performance. We emphasize that there is a lack of a straightforward translation of the I/O bottlenecks

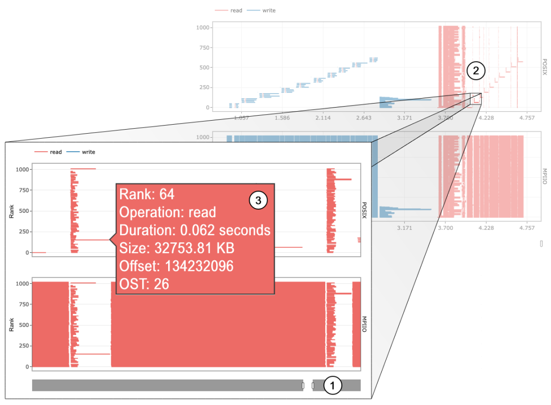

Figure 2: Features available in the interactive visualization of Darshan DXT traces: ① focus on time-intervals, ② zoom into a subset of ranks, and ③ check I/O information.

into potential tuning options. In this paper, we perform a case study using four I/O kernels and demonstrate how the interactive analysis paves the path to performance tuning, albeit a manual effort. The remainder of this section briefly describes a few optimizations that can be used in combination (often misconfigured or disregarded by domain scientists and those with domain knowledge) to improve I/O performance of issues observed in HDF5 I/O kernels using DXT Explorer.

A. High-level I/O Library Optimizations

1) HDF5 Collective Metadata Operations: In HDF5, writing and reading HDF5 internal metadata that describes HDF5 objects can result in many small I/O from some or all MPI ranks, and may cause performance slowdown. To avoid all the MPI ranks participate in the frequent and costly metadata operations, the HDF5 library introduced two API calls that perform metadata writes and reads collectively: `H5Pset_coll_metadata_write` and `H5Pset_all_coll_metadata_ops`. These calls allow HDF5 to perform metadata operations by one rank [16].In the DXT Explorer plots, we observe this issue when MPI ranks are performing small I/O requests (often less than a few hundreds of bytes) to the offset where HDF5 metadata is stored.

2) HDF5 Data Alignment: Parallel File systems typically allocate locks along specific boundaries, aligning file access to the file system stripe size can improve performance. HDF5 allows tuning the alignment properties using the `H5Pset_alignment` function, which comprises of a threshold and an alignment values. By default, HDF5 does not set the alignment. In the interactive plots, we observe this issue with unaligned I/O calls having a high latency.

3) Deferred HDF5 Metadata Flush: The HDF5 library manages a per-file level cache of HDF5 internal metadata and avoids frequent small I/O operations. HDF5 allows the application to control when the entries are flushed or evicted from the cache. The default cache setting may lead to many small I/O requests when the application is operating on a large number of objects. In the DXT Explorer plots, we find this issue when small I/O request appear between data object writes. Adjusting the HDF5 metadata cache configuration [17] to increase the cache size and defer the metadata flush time until file close can significantly improve I/O performance.

B. MPI-IO Level Optimizations

If a group of MPI ranks knows which parts of a file each one is accessing, it becomes possible to merge these requests into a smaller number of larger and more contiguous accesses that span over a large portion of the file. When applied at the client level, this optimization is described as two-phase I/O [6] with collective buffering and data sieving. In our interactive plots, we can observe all the ranks performing I/O requests when these optimizations are not set.

1) Collective Buffering: Collective buffering seeks to reduce I/O time by making all file access large and contiguous, even though it might require additional communication between the processes. ROMIO exposes two user-defined knobs that can control the application of this technique: the number of processes that actually issue the I/O requests in the I/O phase (`cb_nodes`), often referred to as aggregators; and the maximum buffer size on each process (`cb_buffer_size`).

2) Data Sieving: Data sieving aims to reduce I/O latency by making as few requests to the PFS as possible. For read operations, when a process issues non-contiguous requests, instead of reading each piece of data separately, ROMIO reads a single contiguous chunk. ROMIO provides two user-defined parameters to control this optimization (`ind_rd_buffer_size` and `ind_wr_buffer_size`) [6].

C. Parallel File System Level Optimizations

1) File Striping: In a PFS, a file is often partitioned into a sequence of equal-sized data blocks. In Lustre, each block is distributed across a number of OSTs in an operation called data striping. Unlike GPFS, Lustre allows users to configure stripe size and count to improve concurrency in accessing the file system. On Cori, the default stripe count is 1 and the stripe size is 1 MB. DXT Explorer can show the OST that an I/O request accesses on Lustre.

2) Large block IO in GPFS: In GPFS, large files are divided into equal-sized blocks and placed on different disks following a round-robin approach. We observed setting this large block IO parameter for GPFS [18] improving performance when combined with HDF5 metadata optimizations.

IV. Evaluation of Scientific I/O Workloads

In this section, we present a case study of using interactive exploration for identifying I/O performance bottlenecks and tuning performance using the optimizations described in §III-C2. We study four scientific I/O kernels: FLASH, OpenPMD, E2E benchmarks, and block-cyclic I/O. We ran each of these kernels with different configurations more than 5 times and show the best performing run.

A. FLASH-IO

FLASH-IO is an I/O benchmark that simulates the I/O behavior of FLASH [19] code. FLASH has HDF5 and PnetCDF output formats and we focus on the HDF5 backend. We configured FLASH-IO to write 250 3D datasets along with runtime parameter metadata to a single HDF5 file per checkpoint, and three datasets to a plot file. Each FLASH-IO run outputs 2 checkpoint files and 2 plot files. Our experiments on Summit used 64 compute nodes, with 6 ranks per node, and a total of 384 MPI ranks. Each checkpoint file is $\approx$ 2.3 TB, and each plot file is $\approx$ 14 GB.

Figure 3 illustrates a snapshot of the I/O behavior of the baseline FLASH-IO execution on Summit, which uses the default spectrum-mpi/10.3.1.2-20200121 MPI module and HDF5 1.12.1. Using the interactive DXT Explorer plots, we found that the default MPI module does not perform collective I/O operations as expected, depicted as each MPI rank is writing its own data ranging from 12.1 MB to 24.3 MB independently at the POSIX level (excluding the small HDF5 metadata writes). Enabling collective I/O using ROMIO hints with 1 aggregator

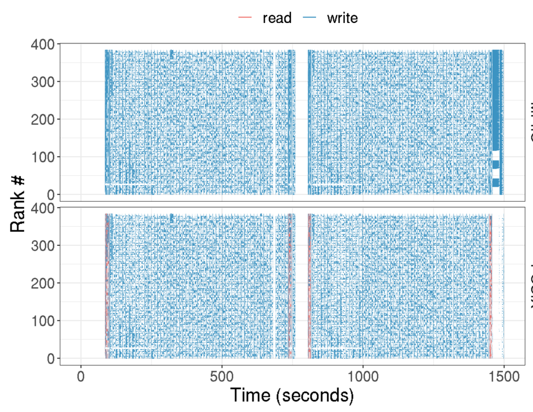

Figure 3: A DXT Explorer snapshot for original FLASH-IO.

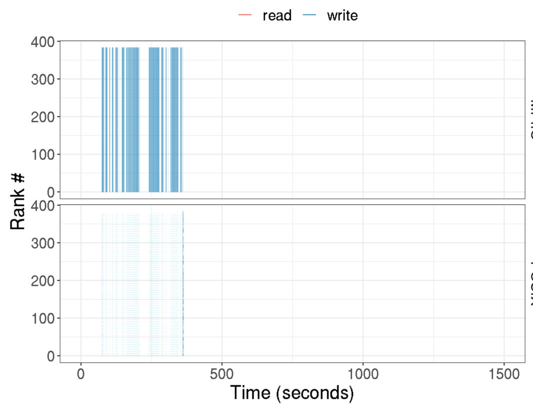

Figure 4: A DXT Explorer snapshot for optimized FLASH-IO.

per node and 16 MB collective buffer size provides $3.2\times$ speedup (§III-B1). Notice the distinct behavior in POSIX and MPI-IO layers in Figs. 3 and 4. Setting the HDF5 alignment size to 16 MB provides an additional $1.18\times$ speedup (§III-A2). Finally, deferring the HDF5 metadata flush provides another $1.1\times$ speedup (§III-A3). Overall, we observed a $4.1\times$ speedup ($1495s$ in Fig. 3 vs. $361s$ in Fig. 4) to run FLASH-IO, and $7.9\times$ speedup to write a checkpoint file ($655s$ vs. $82s$).

B. OpenPMD

OpenPMD [20] is an open meta-data schema targeting particle and mesh data from scientific simulations and experiments. The OpenPMD [21] library provides a reference implementation of the openPMD-standard for file formats such as HDF5 [10], ADIOS [22], and JSON [23]. In this work, we focus on the HDF5 backend using openPMD-api 0.14.1.

On Summit we used 64 compute nodes, 6 ranks per node, and a total of 384 processes. The total file size is $\approx$ 121GB, with no compression set at the HDF5 level. We configured the kernel to write a few meshes and particles in 3D. The meshes are viewed as a grid of dimensions [$64\times32\times32$] of mini blocks whose dimensions are [$64\times32\times32$]. Thus, the actual mesh size is [$65536\times256\times256$]. The kernel runs for 10 iteration steps. In Figure 5, we illustrate the baseline, which uses spectrum-mpi/10.4.0.3-20210112 MPI and HDF5 develop version. The baseline runs on $110.6s$ (avg. of 5 runs). When we enable the `romio_cb_write` and `romio_ds_write` hints and use a collective buffer of 16 MB with 1 aggregator per node (64 total) the runtime drops to $71.60s$, $1.54\times$ faster. Using GFPS' large block I/O combined with collective HDF5 metadata operations, makes the application run in $18.71s$, $3.8\times$ speedup on top of the last optimization. Investigating the DXT Explorer visualization, we also noticed that collective HDF5 metadata were not actually collective due to an issue introduced in HDF5 1.10.5. When using HDF5 1.10.4, the runtime of the combined optimizations is $16.1s$, a $6.8\times$ speedup.

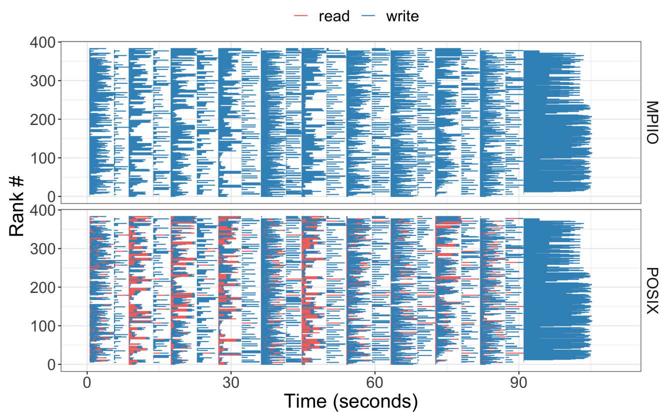

Figure 5: Best run of baseline OpenPMD write in Summit.

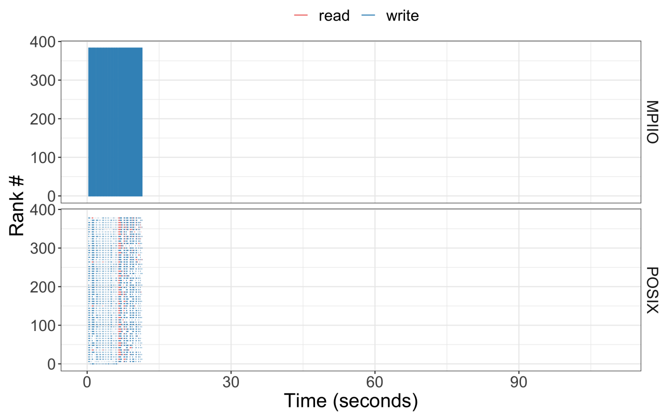

Figure 6: Best execution of OpenPMD after tuned for Summit.

C. 3D Domain Decomposition Kernel

The end-to-end (E2E) I/O kernels proposed by Lofstead et at. [24] are based on domain decomposition problems. We evaluate the 3D domain decomposition pattern with NetCDF4 in this suite that uses HDF5 internally, which was reported to perform very poorly. On Cori, we used 64 compute nodes, with 16 ranks per node, and a total of 1024 MPI ranks. The E2E kernel was configured so each process is represented by $32\times32\times32$ double precision floating points and the sizes represent the number of doubles in each dimension. The 1024 processes are arranged in a $32\times32\times16$ (x by y by z) distribution. This setup generates a $\approx$ 41 GB file. Based on NERSC file striping

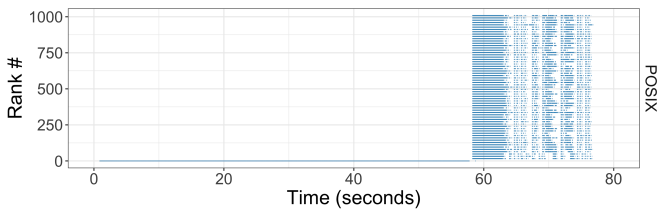

Figure 7: Best execution of the original application behavior, in Cori, where a significant time is spent by rank 0 sequentially writing fill values to all variables when they are defined.

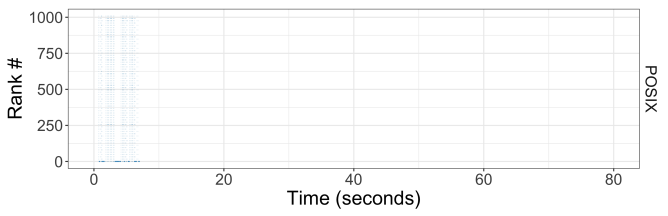

Figure 8: I/O behavior at the POSIX layer of the best execution after setting the `NOFILL` option in Cori.

recommendations[2], for a single shared filed between $10-100$ GB one should set Lustre stripe count to 24 and keep the default stripe size of 1 MB.

Figure 7 shows the write calls at POSIX level. This baseline execution, without any tuning, runs in $80s$ (an average of $85.7s$ over 5 runs). In the plot, we can see that 70% of the time is taken by rank 0 sequentially writing something in the file. Looking at the application's code, rank 0 is filling values to all of the defined variables (10 in this workload). In this initial phase, it issues over 40 thousand write requests with median size of 1 MB. After explicitly disabling the data filling behavior for each one of the 10 variables (i.e., calling `nc_def_var_fill()` with the `NC_NOFILL` in NetCDF4), we achieved a speedup of $10\times$. Figure 8 shows the runtime of $8s$ for this optimized version (avg. of $13.8s$ in 5 runs). On Summit, the same configuration with `NC_NOFILL` optimization achieved $8\times$ improvement from the baseline (runtime reduction from $15.93s$ to $1.97s$).

D. Block-cyclic I/O Pattern

The most commonly encountered data distribution pattern leveraged in distributed memory linear algebra libraries, such as ScaLAPACK [25], ELPA [26] and SLATE [27], is the 2D block-cyclic format, which partitions the matrices into tiles of a fixed dimension and distributes them round robin on a 2D process grid [25]. The block-cyclic data format may be completely described by 6 integers: the number of process rows (n_r) and columns (n_c) which describe the process grid, the total number of rows (m) and columns (n) of the distributed matrix, and the dimension of the row (m_b) and column (n_b) blocking factors which define the local tile dimensions.

As the memory requirement for individual matrices can exceed the capacity of a single compute node, it is desirable to develop procedures which allow for the direct population of block-cyclic data structures from disk. HDF5 offers an attractive API for such developments through the (unioned) selection of block-strided hyperslabs for the population of memory-contiguous data structures from non-contiguous locations on disk.

[2]https://docs.nersc.gov/performance/io/lustre

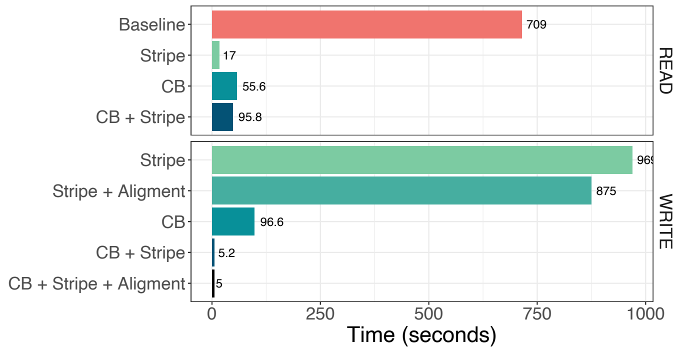

Figure 9: Runtime for the block-cyclic I/O kernel on Cori.

The proxy application developed for this work examines the I/O behaviour for a square matrix with $m = n = 81250$ with FP64 data (~50 GiB). Results are presented for block-cyclic data structures with $m_b = n_b = 128$ with 1024 processes arranged in a $n_r = n_c = 32$ process grid. By using the DXT Explorer, we have applied Lustre striping (§III-C1), MPI-IO collective buffering (§III-B1), and HDF5 alignment optimizations (§III-A2). As shown in Figure 9, using a combination of all the optimizations achieves $41\times$ speedup for reads over the baseline I/O performance. For writes, the baseline reached a time limit of 8 hours. Still, compared to just applying striping (1 MB stripe over 128 OSTs), we observed a $193\times$ speedup by combining it with collective buffering and HDF5 data alignment.

V. Conclusion

In this paper, we proposed an interactive process of identifying I/O performance bottlenecks, mapping them to potential problems, and applying the tuning options to improve I/O performance. We presented a case study with four different I/O kernels with distinct demands and behaviors that exercise various I/O tuning parameters and benefit from existing optimizations. Our *DXT Explorer* (`github.com/hpc-io/dxt-explorer`) tool adds an interactive component to Darshan trace analysis that could aid researchers, developers, and end-users to visually inspect their applications' I/O behavior, zoom-in on areas of interest and have a clear picture of where is the I/O problem.

As evidenced in tuning the four use cases above, applying the available optimizations simply does not guarantee higher I/O performance. Furthermore, the set of techniques used in one application running in one system does not necessarily translate into a fixed rule (neither for the workload nor for the system). In this study, we targeted the gaps between collecting the data to visualize what the application is doing, identifying the bottlenecks, and re-shaping its I/O behavior to perform better in the system. While our tuning using interactive and iterative I/O performance analysis moves a step towards connecting the dots between bottleneck detection and tuning, we note that this is still a tedious process. The gaps of better reporting and automatically mapping performance problems to tuning options and the tools and models required for such mapping need further R&D. Finally, we provide an online companion website with the interactive plots and additional experiments located at `jeanbez.gitlab.io/pdsw-2021`.

Acknowledgment

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 used resources of the National Energy Research Scientific Computing Center under Contract No. DE-AC02-05CH11231, the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory under Contract No. DE-AC05-00OR22725. This work was supported in part by the U.S. Department of Energy, Office of Science, Advanced Scientific Computing Research, under Contract DE-AC02-06CH11357.

References

[1] Y. Li, K. Chang, O. Bel, E. L. Miller, and D. D. E. Long, "CAPES: Unsupervised Storage Performance Tuning Using Neural Network-Based Deep Reinforcement Learning," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '17. New York, NY, USA: ACM, 2017.

[2] M. Agarwal, D. Singhvi, P. Malakar, and S. Byna, "Active Learning-based Automatic Tuning and Prediction of Parallel I/O Performance," in *2019 IEEE/ACM Fourth International Parallel Data Systems Workshop (PDSW)*, 2019, pp. 20–29.

[3] B. Behzad, S. Byna, Prabhat, and M. Snir, "Optimizing I/O Performance of HPC Applications with Autotuning," *ACM Trans. Parallel Comput.*, vol. 5, no. 4, Mar. 2019.

[4] T. Patel, S. Byna, G. K. Lockwood, and D. Tiwari, "Revisiting i/o behavior in large-scale storage systems: The expected and the unexpected," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '19. New York, NY, USA: Association for Computing Machinery, 2019.

[5] B. Xie, H. Tang, S. Byna, J. Hanley, Q. Koziol, T. Li, and S. Oral, "Battle of the Defaults: Extracting Performance Characteristics of HDF5 under Production Load," in *CCGrid 2021*, 2021.

[6] R. Thakur, W. Gropp, and E. Lusk, "Data sieving and collective i/o in romio," in *Proceedings. Frontiers '99. Seventh Symposium on the Frontiers of Massively Parallel Computation*, 1999, pp. 182–189.

[7] B. Behzad, H. V. T. Luu, J. Huchette, S. Byna, R. Aydt, Q. Koziol *et al.*, "Taming parallel I/O complexity with auto-tuning," in *SC'13: Proceedings of the International Conf. on High Performance Computing, Networking, Storage and Analysis*. ACM, 2013, pp. 1–12.

[8] R. McLay, D. James, S. Liu, J. Cazes, and W. Barth, "A user-friendly approach for tuning parallel file operations," in *SC '14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '14. Piscataway, NJ, USA: IEEE Press, 2014, pp. 229–236.

[9] P. Carns, K. Harms, W. Allcock, C. Bacon, S. Lang, R. Latham, and R. Ross, "Understanding and improving computational science storage access through continuous characterization," *ACM Trans. Storage*, vol. 7, no. 3, Oct. 2011. [Online]. Available: https://doi.org/10.1145/2027066.2027068

[10] The HDF Group. (1997-) Hierarchical Data Format, version 5. [Online]. Available: http://www.hdfgroup.org/HDF5

[11] S. Byna, M. Breitenfeld, B. Dong, Q. Koziol, E. Pourmal, D. Robinson, J. Soumagne, H. Tang, V. Vishwanath, and R. Warren, "ExaHDF5: Delivering Efficient Parallel I/O on Exascale Computing Systems," *JCST*, vol. 35, pp. 145–160, 2020.

[12] C. Xu, S. Snyder, O. Kulkarni, V. Venkatesan, P. Carns, S. Byna, R. Sisneros, and K. Chadalavada, "DXT: Darshan eXtended Tracing," 1 2019. [Online]. Available: https://www.osti.gov/biblio/1490709

[13] C. Wang, J. Sun, M. Snir, K. Mohror, and E. Gonsiorowski, "Recorder 2.0: Efficient parallel I/O tracing and analysis," in *2020 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, May 2020, pp. 1–8.

[14] H. Wickham, *Ggplot2: Elegant graphics for data analysis*, 2nd ed., ser. Use R! Cham, Switzerland: Springer Int. Publishing, Jun. 2016.

[15] P. T. Inc. (2015) Collaborative data science. Montreal, QC. [Online]. Available: https://plot.ly

[16] "HDF5 Collective Metadata I/O Documentation." [Online]. Available: https://support.hdfgroup.org/HDF5/docNewFeatures/NewFeaturesCollectiveMetadataIoDocs.html

[17] "HDF5 Fine-tuning the Metadata Cache Documentation." [Online]. Available: https://support.hdfgroup.org/HDF5/docNewFeatures/NewFeaturesFineTuningMetadataCacheDocs.html

[18] J.-P. Prost, R. Treumann, R. Hedges, B. Jia, and A. Koniges, "MPI-IO/GPFS, an Optimized Implementation of MPI-IO on Top of GPFS," in *Proceedings of the 2001 ACM/IEEE Conference on Supercomputing*, ser. SC '01. New York, NY, USA: ACM, 2001, p. 17.

[19] A. Dubey, K. Antypas, A. Calder, B. Fryxell, D. Lamb, P. Ricker, L. Reid, K. Riley, R. Rosner, A. Siegel *et al.*, "The software development process of flash, a multiphysics simulation code," in *2013 5th International Workshop on Software Engineering for Computational Science and Engineering (SE-CSE)*. IEEE, 2013, pp. 1–8.

[20] Huebl, Axel and Lehe, Remi and Vay, Jean-Luc and Grote, David P. and Sbalzarini, Ivo F. and Kuschel, Stephan and Sagan, David and Mayes, Christopher and Perez, Frederic and Koller, Fabian and Bussmann, Michael. (2015) openPMD: A meta data standard for particle and mesh based data. [Online]. Available: https://doi.org/10.5281/zenodo.1167843

[21] Koller, Fabian and Poeschel, Franz and Gu, Junmin and Huebl, Axel. (2019) openPMD-api 0.10.3: C++ & Python API for Scientific I/O with openPMD. DOI: 10.14278/rodare.209. [Online]. Available: https://doi.org/10.14278/rodare.209

[22] J. F. Lofstead, S. Klasky, K. Schwan, N. Podhorszki, and C. Jin, "Flexible IO and integration for scientific codes through the adaptable IO system (ADIOS)," in *CLADE*. NY, USA: ACM, 2008, pp. 15–24.

[23] F. Pezoa, J. L. Reutter, F. Suarez, M. Ugarte, and D. Vrgoč, ""foundations of json schema"," in *Proceedings of the 25th International Conference on World Wide Web*. International World Wide Web Conferences Steering Committee, 2016, pp. 263–273.

[24] J. Lofstead, M. Polte, G. Gibson, S. Klasky, K. Schwan, R. Oldfield, M. Wolf, and Q. Liu, "Six degrees of scientific data: Reading patterns for extreme scale science io," in *Proceedings of the 20th International Symposium on High Performance Distributed Computing*, ser. HPDC '11. New York, NY, USA: ACM, 2011, p. 49–60.

[25] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet *et al.*, *ScaLAPACK users' guide*. SIAM, 1997.

[26] A. Marek, V. Blum, R. Johanni, V. Havu, B. Lang, T. Auckenthaler, A. Heinecke, H.-J. Bungartz, and H. Lederer, "The ELPA library: scalable parallel eigenvalue solutions for electronic structure theory and computational science," *Journal of Physics: Condensed Matter*, vol. 26, no. 21, p. 213201, 2014.

[27] M. Gates, A. Charara, J. Kurzak, A. YarKhan, M. A. Farhan, D. Sukkari, and J. Dongarra, "SLATE Users Guide," University of Tennessee, Tech. Rep., 2020.

Appendix

We provide a **companion website** with additional *DXT Explorer* snapshots and the interactive plots of this paper:

```
jeanbez.gitlab.io/pdsw-2021
```

Our companion repository also showcases two complementary results: OpenPMD's in Cori and E2E in Summit. For OpenPMD in Cori, we saw a 77% performance improvement from the baseline by combining MPI-IO two-phase I/O, collective metadata operations in HDF5, deferred metadata writes, and paged allocation. E2E in Summit demonstrated similar behavior as in Cori (a lot of the time is taken by rank 0 sequentially writing fill values to all 10 variables. Our website provides an interactive visualization of the baseline and optimized version after explicitly disabling the data filling behavior.