Initial Experiences with DAOS Object Storage on Aurora

Rob Latham*, Robert B. Ross*, Philip Carns*, Shane Snyder*,
Kevin Harms†, Kaushik Velusamy†, Paul Coffman†, Gordon McPheeters†
*Mathematics and Computer Science Division, †Argonne Leadership Computing Facility
Argonne National Laboratory, Lemont, IL, USA 60439
{robl,rross,carns,ssnyder}@mcs.anl.gov, harms@alcf.anl.gov, {kaushik.v,pcoffman,gmcpheeters}@anl.gov

***Abstract*—The storage subsystem of the Aurora platform at Argonne National Laboratory offers the potential for unprecedented application I/O performance. Its hardware stack combines NVMe drives, persistent memory devices, a large collection of storage servers, and a unified RDMA-capable network fabric. Its software stack is based on the DAOS object storage system, thereby putting into practice decades of I/O research into minimizing overhead, maximizing access concurrency, and streamlining application interfaces. Taken together, these elements combine to present up to 230 PiB of projected storage capacity and up to 31 TiB/s of projected I/O throughput to applications executing on Aurora.**

This storage platform is unique, however, and presents a new set of challenges to practitioners. To unlock the full potential of this system, we must revisit fundamental issues such as hardware topology, network concurrency, and I/O APIs to understand their impact on real-world I/O performance. In this paper we present our initial experiences with the DAOS storage system on Aurora and characterize its sensitivity to these parameters at small scale. We discuss our initial impressions and offer recommendations for how to extract maximum effective performance from this class of storage system.

I. Introduction

The rise of data-driven science and artificial intelligence (AI) methods has driven the demand for flexible large-scale storage systems that can effectively service a diverse range of I/O workloads. Storage technologies have not improved in their capabilities at the same rate as compute. However, high-performance computing (HPC) networking coupled with nonvolatile memory technologies can provide the basis for a high-performance storage system. The degree to which that performance is exploitable by users depends on the software that is used to organize that hardware. Historically parallel file systems have been performant for only a narrow range of application use cases.

Object storage is an umbrella term, but in general it refers to storage systems that eschew conventional hierarchical file and directory representations for a flat namespace of uniquely addressable objects, each with its own permissions and attributes. Object storage provides new opportunities for performance optimization by (a) increasing concurrency by accessing independent objects with minimal contention, (b) avoiding legacy POSIX APIs that are inefficient for parallel access [1], [2], and (c) taking advantage of interfaces that expose more information about the underlying data layout.

In this paper we present our initial experiences in evaluating the performance characteristics of the DAOS object storage system on the Aurora platform at Argonne National Laboratory at small scale. I/O performance depends on utilizing Aurora nodes, networking, and storage effectively. As we introduce each component of the technology stack, we will point out both pitfalls and opportunities as we work toward maximizing application I/O.

II. Background and Related Work

In addition to the novel DAOS architecture, the Aurora supercomputer possesses other features that impose new challenges for I/O performance tuning. The compute nodes have a complex topology with two many-core CPUs, six GPUS, and a complex PCI bus. Each is connected to the network fabric by eight network interface cards. Although the hardware is exceptionally powerful, the full performance is available only if it can be utilized with a high degree of concurrency.

A. Object stores in HPC

Object storage architectures are a long-standing building block for parallel file systems including Panasas [3], Ceph [4], and PVFS [5]. Research efforts have shown the potential benefits of exposing object APIs directly to applications [6], [7]. Prior to the introduction of DAOS, however, no production HPC storage systems have pursued this approach, largely because of the inertia of legacy data models and a lack of consensus on appropriate object storage semantics for HPC applications. Object storage models are much more widely used in cloud computing environments,[1] but these deployments emphasize restful, web-service-oriented interfaces and highly distributed but sequential access patterns. They are unsuitable for HPC deployment because of the need for HPC applications to access large datasets in parallel within a single coherent system.

B. DAOS

The DAOS storage system [8] is a distributed storage system developed by Intel Corporation and the DAOS Foundation to apply the concepts of object storage in large-scale computing environments. Its API presents a data model consisting of objects (replicated or erasure-coded data units with key-value

[1] https://aws.amazon.com/s3/

979-8-3503-5554-3/24/$31.00 ©2024 IEEE
DOI 10.1109/SCW63240.2024.00171

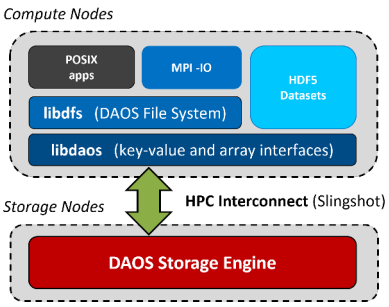
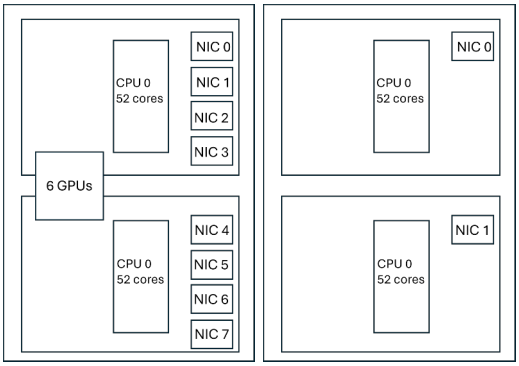


Fig. 1: DAOS provides multiple interfaces to storage.



(a) Compute Node (b) Storage Node

Fig. 2: High-level schematic of an Aurora compute node (left) and an Aurora storage node (right). Aurora compute nodes consist of two 52-core CPUs and 6 GPUs. These nodes are attached to the Slingshot interconnect via four dual-ported Slingshot-11 network cards. Storage nodes by contrast have no GPUs and two network cards.

access methods), containers (logical groupings of objects), and pools (provisioned storage capacity). The DAOS API is implemented as a user-level library that can in turn be directly mapped to a variety of application-level data models while avoiding costly translation and can be mapped directly to high-level application libraries without using costly POSIX operating system abstractions. Internally, DAOS is designed to minimize overhead for access to high-performance remote hardware. It includes native support for Remode Direct Memory Access (RDMA) network transports, persistent memory devices, and Non-Volatile Memory Express devices. Both the DAOS server daemons and client library operate exclusively in user space with minimal context switching in order to reduce software overheads that would otherwise become the limiting factor in performance in this environment. The DAOS model is especially well suited to HPC because it can be deployed directly within the HPC compute node network fabric; its software architecture is amenable to multi-tenant, highly concurrent access patterns; and its API can be readily mapped to a diverse range of application-level I/O libraries.

DAOS breaks with the traditional POSIX interface, offering a wide array of interfaces to storage, as shown in Figure 1. Applications can access DAOS through POSIX without modification through compatibility libraries. The ROMIO MPI-IO implementation [9] uses the libdfs interface directly. HDF5 [10] stores and retrieves its information through DAOS's lowest-level key-value and array interfaces.

Applications have a "no change" path to DAOS via a "FUSE"-based approach [17]. This path provides the traditional POSIX I/O interface that applications have used for decades: every I/O and metadata operation routes through FUSE (though as we will see in Section IV-E, this simplicity comes with a performance cost). DAOS also provides an interception library called `libioil`, which intercepts only I/O calls, leaving metadata operations to the FUSE-based approach. An alternative interception library, `libpil4dfs`, captures some common metadata operations in addition to I/O. For developers of applications or libraries wishing to target DAOS directly and willing to write code targeting a new programming interface, there is a file-system-like library interface called `libdfs` (sometimes known as DFS). Programs using this interface entirely bypass the kernel and operate directly on the storage system.

C. DAOS studies

DAOS on Aurora has seen some early performance evaluations for IO500. [2] Our evaluations in this work seek to further understand how DAOS behaves under a variety of workloads.

Some applications developers are excited to target DAOS's novel interfaces. The European Center for Medium-Range Weather Forecasts has started porting components of its workflow to DAOS to enable higher concurrency [11]. The HDF5 library provides its familiar multidimensional array interface on top of DAOS's lower-level APIs [12], which should further ease applications porting to this new storage system. In this work we focus on the tuning and platform-specific features of DAOS that adopters might need to keep in mind.

III. System Environment and Architecture

The Aurora system at the Argonne Leadership Computing Facility consists of 10,624 compute nodes. In Figure 2a we show an approximate schematic depicting one of those nodes: two 52-core CPUs (Intel Xeon Sapphire Rapids) are paired with six Intel Ponte Veccio GPUs. Eight network links—two 2-port cards per CPU—attach these nodes to the Slingshot-11 network. The Slingshot network carries both internode communication traffic and storage traffic. Each network card advertises 25 GiB/s of bandwidth or 200 GiB in aggregate. In later sections we will see how close we can come to that "speed of light" figure.

The DAOS servers (Figure 2b) run on slightly different hardware. The servers have no GPUs and only one network link per CPU. We can compute a rough estimate of DAOS performance—2 network cards per server, 20 GiB/s per card yields 40 GiB/s per server.

[2] https://io500.org/submissions/view/695

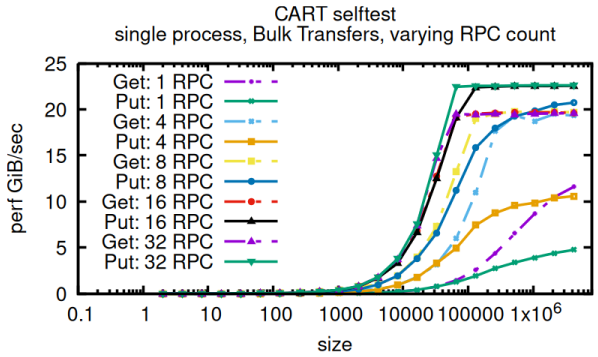


Fig. 3: Single node, single process; evaluating effect of RPC count on performance

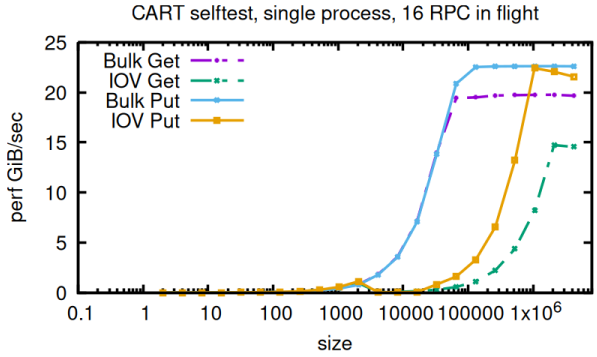


Fig. 4: Single node, single process; 16 RPCs in flight

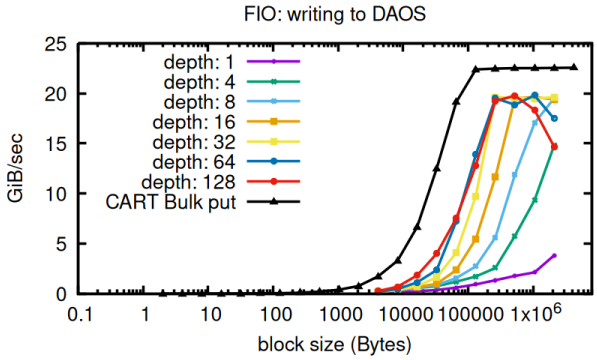


Fig. 5: Impact of I/O size and number of concurrent operations on single-client DAOS performance

The Aurora system is still undergoing acceptance and shakedown. We carried out the tests and evaluations for this early experience paper on a DAOS pool supported by 20 DAOS providers (one provider per node), giving a rough peak bandwidth estimate of 800 GiB/s. Inside that pool we created a container in "SX" mode: use all DAOS providers with no redundancy

IV. Performance Evaluation

To understand this unique storage system, we need to rebuild our personal mental models of how a parallel storage system behaves. To that end, we evaluated DAOS from the bottom up, characterizing network, single-node, and parallel accesses. Space constraints prevent us from evaluating random I/O and metadata workloads, two workloads of high interest to the AI community.

A. Single-process network performance

How do concurrency and transfer size impact the performance of the Slingshot network powering DAOS? DAOS uses the Mercury [13] remote procedure call (RPC) layer, managing operations with a library called CART [14]. We can measure network performance with CART's `self_test` tool. This tool behaves like DAOS but without any storage operations, thus allowing us to evaluate the network communication of a DAOS client communicating to DAOS servers.

As discussed, Aurora nodes contain eight network cards. In this section we exercise one card but use insights from these tests to build up our performance models. The throughput of sequential network operations is inherently constrained by protocol handshaking and memory access overhead. Increasing operation concurrency enables these tasks to be overlapped so that the network link is saturated with pipelined traffic at all times. Our first question, therefore, is how much operation concurrency is necessary to achieve maximum performance per network card.

In Figure 3 we used `self_test` to evaluate the performance of RDMA transfers between clients and DAOS servers while varying request size and concurrency. An RPC count of 1 means clients issue requests sequentially, waiting for a response for each before proceeding. An RPC count of 32 means clients keep 32 requests in flight at all times. We see performance converging for both RMA put and get operations with an RPC count of 16, with some smaller gains with higher RPC counts. Peak performance is achieved with a request size of 64 KiB.

Should we consider eager messaging optimizations? Mercury and DAOS have two message modes. The "IOV" mode is analogous to MPI eager mode messaging—the payload is sent along with the initial message request. In "bulk" mode, the initiator informs the target about a large RMA transfer; then the target performs a put (write) or get (read) operation. In Figure 4 we see bulk mode outperforming all but the smallest transfers. With this result we can confirm DAOS's choice of "always RMA" was the right one for this platform.

B. Single-process end-to-end performance

What impact does storage have on levels of concurrency and sizes of transfers required to achieve highest performance? The widely used `fio` benchmark [15] can generate a variety of I/O workloads. It can also post multiple operations asynchronously, which as we saw in Section IV-A greatly impacts performance. The benchmark has an "engine" for DAOS that

uses the libdfs interface directly so that we can isolate DAOS storage performance from application API considerations.

In Figure 5 we measure single-node, single-process performance as we vary both the transfer size and the number of concurrent operations. We adjust the DAOS "chunk size" to match the transfer size. To give the numbers some additional context, we also plot a network-only CART self_test experiment. As expected, a concurrency depth of 1 is unable to drive any performance from this client. It takes at least 32 concurrent operations to reach the performance plateau. We also see that request sizes of 256 KiB or larger are needed. This is in contrast with our findings in the network-only testing of the preceding subsection; the introduction of storage device access raises the level of request concurrency (16 to 32) and request size (64 KiB to 256 KiB) needed to saturate performance for a single client process on a single network link. We also now observe that we can have too much concurrency as well: note that performance drops off when issuing 128 concurrent requests of 1 MiB or more.

C. Multiple-process end-to-end performance

How does node topology impact performance as we add more on-node concurrency? To study performance from multiple clients on a single node, we need a multiclient benchmark, the multiprocess MPI-based `IOR` benchmark [16]. In Figure 2 we note that four network cards are connected to one CPU and four to the other. This configuration suggests that we must consider the mapping of processes and cores to network cards in order to use the network cards effectively. Process placement can have a large impact on observed performance.

We will go into more detail about our IOR benchmarks in the next sections. For now it is sufficient to know that we are using IOR to measure DAOS performance as we vary the number of simultaneous clients on a node. In Figure 6a and Figure 6b we run the same benchmark with different CPU binding strategies. The "no binding" case uses the default process placement. It schedules up to 52 tasks on one CPU before scheduling on the second CPU. The "bucket" and "breadth" strategies schedule across both CPUs. In the bucket case processes are distributed across both CPUs evenly but depth first: the first half of the processes go to CPU0, and the second half go to CPU1. In the "breadth" case, processes are scheduled round robin over both CPUs. We also consider a fourth case that takes the "breadth" approach but starts on the fifth core of CPU0 on the theory that the first four cores might be occupied by operating system tasks and network card interrupts.

Most important, the default "no binding" case will be able to deliver high performance only if MPI processes are running on every core. With the network cards split up between NUMA domains, Aurora jobs will have to be mindful to bind processes across both domains. Any binding strategy that distributes processes across both CPUs (and, in turn, network cards) will yield improved performance. We continue to investigate the impact of binding strategies on particular workloads. Unless stated otherwise, the remainder of experiments are going to use the "breadth skip" binding strategy.

D. Maximizing aggregate single-node performance

Do these concurrency and blocksize observations still hold in an MPI-based application? The IOR benchmark allows us to measure I/O performance through a variety of methods and access patterns. Additionally, IOR implements multiple storage clients. In this experiment we use IOR's DAOS driver. That driver operates directly on the DAOS storage, bypassing any kernel or FUSE interfaces. The libdfs interface will also split large requests into smaller chunks, with a default chunk size of 1 MiB. The library will split up requests larger than the chunk size and post them concurrently.

As established in Sections IV-A and IV-B, to extract peak storage performance from a single Aurora node, we require parallelism across the eight network cards plus multiple operations in flight. We ran an IOR benchmark issuing large contiguous requests (640 MiB block size, 16 MiB transfer size) observing performance as we increased MPI processes. A single MPI process, or even one process per network card, is unable to achieve high performance. The performance curves in Figure 7 do not begin to level off until around 32 processes (4 processes per network card) with a distinct "knee" in the curve at 16 processes.

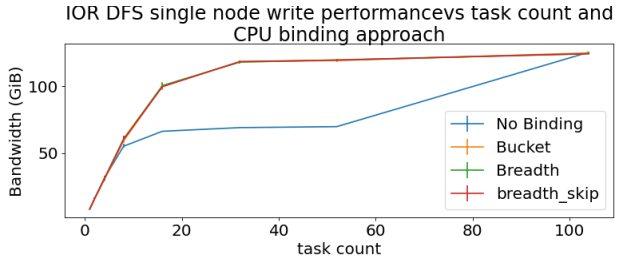
To better understand the impact of transfer size and parallel I/O transfers, we took the 16-process IOR benchmark and varied the DAOS transfer size. In Figure 8 we can see striking variations in performance with smaller transfer sizes. A 512 KiB chunk size provides consistent high performance.

Compare these results with the FIO tests shown earlier in Figure 5. In the IOR case we have more processes and more network cards involved than were used in the FIO tests. Even with increased concurrency (remember, we have placed processes such that the benchmark uses all 8 network cards), performance suffers with the smaller transfer sizes. The rough shape of the performance curves still holds: although the knee is less prominent, best performance happens with transfer sizes 512 KiB and higher. When we looked at FIO, which tests only one client on one network card, the ideal transfer sizes were greater than 256 KiB.

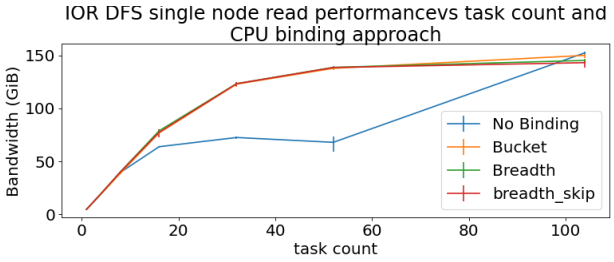
E. Choosing an optimal interface for POSIX access

What impact does an application's choice of interface have on DAOS performance? DAOS provides several interfaces to its storage: FUSE or interception libraries for legacy codes or the `libdfs` file-like programming interface.

In Figure 9 we show how well these interfaces perform on one node of Aurora as we increase the task count. The FUSE approach provides very little performance, for well-known reasons [18]. FUSE serializes all requests through the kernel module and, at best, routes requests through one network card, leaving the other seven interfaces on Aurora idle. Facilities deploy the FUSE interface for legacy applications, not for performance. Interception libraries provide access to DAOS without code changes, requiring only that job scripts set an



(a) IOR writes



(b) IOR reads

Fig. 6: Impact of CPU binding strategies

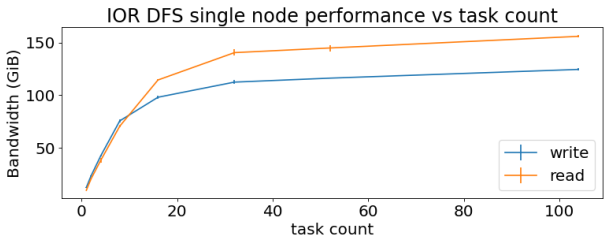


Fig. 7: With careful process placement, 32 processes begin to saturate a single node's connection to DAOS.

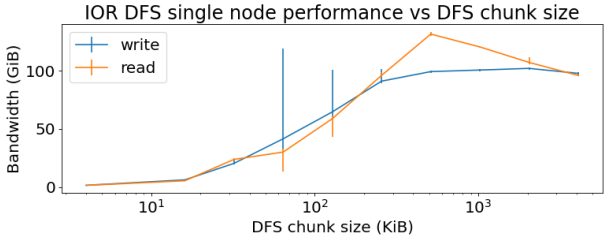


Fig. 8: IOR read and write performance to DAOS using 16 processes (2 processes per network card), evaluated at various transfer sizes

environment variable. The IOR benchmark writes and reads a large amount of data and makes few metadata operations, but routing even this small number of metadata operations through the kernel interface incurs serialization and performance costs. We are pleased to see that in the most recent versions of DAOS the `libpil4dfs` interception library will be the preferred interception library for DAOS and that use of the older `libioil` library will be discouraged.

F. Scaling

What does performance look like when we have multiple nodes involved? In Figure 10 we show the results of weak scaling with an IOR benchmark. We increase both the number of nodes and the tasks per node. Every task is writing 64 MiB in 16 MiB transfer sizes across six segments for four iterations. Our single-node studies in Section IV-D demonstrated the need to use multiple tasks to achieve full bandwidth out of a node. Also, recall from Figure 2 that compute nodes contain more network links than storage nodes.

In Figure 10 we can see that with a large number of clients no on-node parallelism is required. The 256-client benchmark sees little to no improvement as we add more tasks. At smaller client counts, more tasks are required, and of course with a single node no amount of tasks can reach 800 GiB/s.

V. Conclusions and Future Work

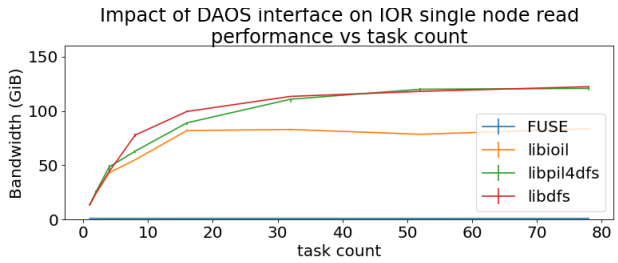
As Aurora continues its shakedown process and prepares for production, we will increase the scale of our concurrency studies. We also want to characterize DAOS's latency, measure the impact of Aurora's Optane caches, evaluate the impact of redundancy and erasure coding, and explore ways that libraries can leverage DAOS's nonblocking and list-io features while presenting simplified interfaces to applications and end users. The benchmarks and evaluations in this paper focus on modeling and simulation workloads: AI workloads are an obvious next area of study.

In this initial study of DAOS on Aurora we have collected some practices that users will have to adopt in order to get the most out of the storage system. Single-process, single-threaded operations will not be able to generate enough concurrency to achieve high performance. Request sizes should be hundreds of kilobytes: smaller requests will perform poorly. The legacy FUSE interface should be avoided at all costs, by using either an interception library or a library such as MPICH or HDF5 that uses a native DAOS interface.

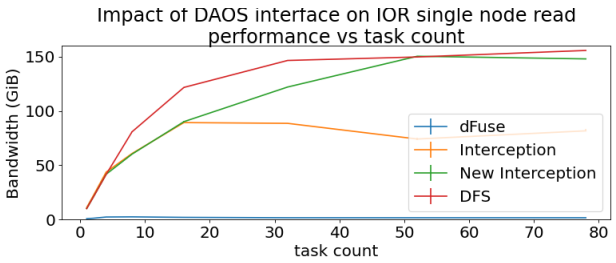
While Aurora's DAOS may be a unique configuration today, we think these tuning approaches are indicative of the kind of performance tuning that will be needed for future generation storage systems; these concerns represent a changing of the guard from times when subtle software overhead and client-side concurrency were not major factors in effective performance.

Acknowledgment

This work was done on a preproduction supercomputer with early versions of the Aurora software development kit. This research used resources of the Argonne Leadership Computing Facility, a U.S. Department of Energy (DOE) Office

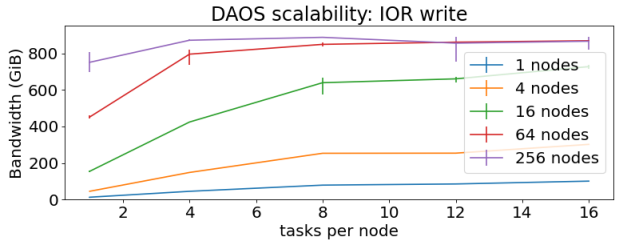


(a) Write

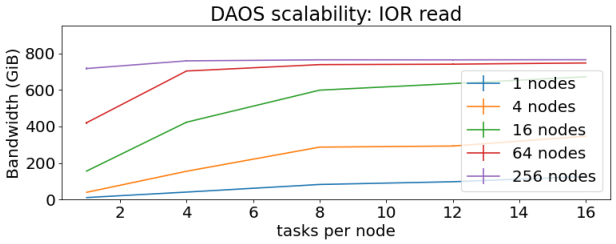


(b) Read

Fig. 9: Choice of interface and impact on I/O performance



(a) Write



(b) Read

Fig. 10: Impact of node and task parallelism on IOR benchmark

of Science user facility at Argonne National Laboratory and is based on research supported by the U.S. DOE Office of Science-Advanced Scientific Computing Research Program, under Contract No. DE-AC02-06CH11357.

REFERENCES

[1] D. Kimpe and R. Ross, "Storage models: Past, present, and future," *High Performance Parallel I/O*, pp. 335–345, 2014.

[2] C. Wang, K. Mohror, and M. Snir, "File system semantics requirements of hpc applications," in *Proceedings of the 30th International Symposium on High-Performance Parallel and Distributed Computing*, ser. HPDC '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 19–30. [Online]. Available: https://doi.org/10.1145/3431379.3460637

[3] B. Welch, M. Unangst, Z. Abbasi, G. A. Gibson, B. Mueller, J. Small, J. Zelenka, and B. Zhou, "Scalable performance of the panasas parallel file system." in *FAST*, vol. 8, no. 2008, 2008, pp. 1–17.

[4] S. Weil, S. A. Brandt, E. L. Miller, D. D. Long, and C. Maltzahn, "Ceph: A scalable, high-performance distributed file system," in *Proceedings of the 7th Conference on Operating Systems Design and Implementation (OSDI'06)*, 2006, pp. 307–320.

[5] R. Latham, N. Miller, R. Ross, P. Carns, and C. Univ, "A next-generation parallel file system for linux cluster." *LinuxWorld Mag.*, vol. 2, 01 2004.

[6] D. Goodell, S. J. Kim, R. Latham, M. Kandemir, and R. Ross, "An evolutionary path to object storage access," in *Proceedings of the 7th Parallel Data Storage Workshop*, Salt Lake City, UT, Nov. 2012.

[7] J. Liu, Q. Koziol, G. F. Butler, N. Fortner, M. Chaarawi, H. Tang, S. Byna, G. K. Lockwood, R. Cheema, K. A. Kallback-Rose, D. Hazen, and M. Prabhat, "Evaluation of hpc application i/o on object storage systems," in *2018 IEEE/ACM 3rd International Workshop on Parallel Data Storage & Data Intensive Scalable Computing Systems (PDSW-DISCS)*, 2018, pp. 24–34.

[8] Z. Liang, J. Lombardi, M. Chaarawi, and M. Hennecke, "Daos: A scale-out high performance storage stack for storage class memory," in *Supercomputing Frontiers*, D. K. Panda, Ed. Cham: Springer International Publishing, 2020, pp. 40–54.

[9] R. Thakur, W. Gropp, and E. Lusk, "On implementing MPI-IO portably and with high performance," in *Proceedings of the Sixth Workshop on Input/Output in Parallel and Distributed Systems*, May 1999, pp. 23–32. [Online]. Available: http://www.mcs.anl.gov/~thakur/papers/mpio-impl.ps

[10] "HDF5," `http://www.hdfgroup.org/HDF5/`.

[11] N. Manubens, S. D. Smart, E. Danovaro, T. Quintino, and A. Jackson, "Reducing the impact of i/o contention in numerical weather prediction workflows at scale using daos," in *Proceedings of the Platform for Advanced Scientific Computing Conference*, ser. PASC '24. New York, NY, USA: Association for Computing Machinery, 2024. [Online]. Available: https://doi.org/10.1145/3659914.3659926

[12] J. Soumagne, J. Henderson, M. Chaarawi, N. Fortner, S. Breitenfeld, S. Lu, D. Robinson, E. Pourmal, and J. Lombardi, "Accelerating hdf5 i/o for exascale using daos," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 4, pp. 903–914, 2022.

[13] J. Soumagne, D. Kimpe, J. Zounmevo, M. Chaarawi, Q. Koziol, A. Afsahi, and R. Ross, "Mercury: Enabling remote procedure call for high-performance computing," in *Proceedings of the IEEE Cluster Conference*, Sep. 2013.

[14] https://github.com/daos-stack/cart.

[15] J. Axboe. (2022) Flexible I/O Tester. [Online]. Available: https://github.com/axboe/fio

[16] Ior. [Online]. Available: https://github.com/hpc/ior

[17] Linux fuse (filesystem in userspace). [Online]. Available: https://github.com/libfuse/libfuse

[18] B. K. R. Vangoor, V. Tarasov, and E. Zadok, "To fuse or not to fuse: performance of user-space file systems," in *Proceedings of the 15th Usenix Conference on File and Storage Technologies*, ser. FAST'17. USA: USENIX Association, 2017, p. 59–72.

Appendix: Artifact Description/Artifact Evaluation

Artifact Description (AD)

I. Overview of Contributions and Artifacts

A. Paper's Main Contributions

This paper performs a detailed performance study of the new DAOS file system and Aurora supercomputer, installed at the Argonne Leadership Computing Facilty.

- C_1 Behavior of the Slingshot interconnect with respect to request size and concurrency
- C_2 Behavior of the DAOS storage system with respect to request size and concurrency
- C_3 Importance of cpu binding and task placement with respect to observed performance.
- C_4 Evaluation of overhead of DAOS programming interfaces
- C_5 Initial small-scale node and task parallellism evaluation.

B. Computational Artifacts

A_1 https://doi.org/10.5281/zenodo.13328336

Artifact ID	Contributions Supported	Related Paper Elements
A_1	C_1	Figures 3-4
A_1	C_2	Figure 5, Figure 8
A_1	C_3	Figures 6-7
A_1	C_4	Figure 9
A_1	C_5	Figure 10

II. Artifact Identification

A. Computational Artifact A_1

Relation To Contributions

This artifact contains all the job scripts, post-processing, and graph-generating tools used to generate the figures in the paper.

Expected Results

The experiments characterize the performance of the major components of the DAOS storage stack on Aurora. By understanding the behavior of the network and storage at smaller scales, we aim to better understand how DAOS will perform at larger scale and with the wide array of scientific applications expected to use it.

Expected Reproduction Time (in Minutes)

The expected computational time to run all experiments in this artifact is about two hours:

- CART self-test: 10 minutes
- FIO blocksizes: 70 minutes
- IOR and cpu binding strategies: 15 minutes
- Impact of API choice on I/O performance: 10 minutes
- small scale IOR runs: 15 minutes

Artifact Setup (incl. Inputs)

Hardware: These tests require access to DAOS storage. Most notably, Aurora compute nodes have eight network cards.

Software: Tests should work with any version of DAOS. We ran against DAOS 2.2 on both clients and servers, though Aurora has upgraded both several times since then. External benchmarks:

- CART self-test: packaged along with DAOS client tools
- FIO https://github.com/axboe/fio, version 3.36
- IOR https://github.com/hpc/ior , git revision b666c40cbc

Datasets / Inputs: All tests used synthetic or self-generated datasets. No need to download external datasets.

Installation and Deployment: Benchmarks are buildable with any standard MPI compiler. On Aurora we used Intel's OneAPI compiler `oneapi/eng-compiler/2023.10.15.002` though these tests make no use of DPC++ features. The MPI implementation was configured with DAOS support, but the IOR experiments in this paper used libdfs directly.

Artifact Execution

We include for each experiment the PBS job script to run the associated tests.

Artifact Analysis (incl. Outputs)

An `extract.sh` shell script can convert the output into something more easily plot-able. The IOR benchmarks use a `ior-extract.sh` script, documented in README.md files.

Once you have post-processed files, the directories contain gnuplot or matplotlib plotting scripts.