An MPI-IO In-Memory driver for non-volatile pooled memory of the Kove XPD

An MPI-IO In-Memory Driver for Non-Volatile Pooled Memory of the Kove XPD

Julian M. Kunkel and Eugen Betke

German Climate Computing Center (DKRZ)

Abstract. Many scientific applications are limited by the performance offered by parallel file systems. SSD based burst buffers provide significant better performance than HDD backed storage but at the expense of capacity. Clearly, achieving wire-speed of the interconnect and predictable low latency I/O is the holy grail of storage. In-memory storage promises to provide optimal performance exceeding SSD based solutions. Kove®'s XPD® offers pooled memory for cluster systems. This remote memory is asynchronously backed up to storage devices of the XPDs and considered to be non-volatile. Albeit the system offers various APIs to access this memory such as treating it as a block device, it does not allow to expose it as file system that offers POSIX or MPI-IO semantics.
In this paper, we 1) describe the XPD-MPIIO-driver which supports the scale-out architecture of the XPDs. This MPI-agnostic driver enables high-level libraries to utilize the XPD's memory as storage. 2) A thorough performance evaluation of the XPD is conducted. This includes scale-out testing of the infrastructure and „metadata" operations but also performance variability.
We show that the driver and storage architecture is able to nearly saturate wire-speed of Infiniband (60+ GiB/s with 14 FDR links) while providing low latency and little performance variability.

Keywords: MPI-IO, evaluation, in-memory storage

1 Introduction

In an alternative storage architecture, a burst buffer [1,2] is placed between compute nodes and the storage. Acting as an intermediate storage tier, it's goal is to catch the I/O peaks from the compute nodes. Therefore, it provides a low latency and high bandwidth to the compute nodes, but also utilizes the backend storage by streaming data constantly at a lower bandwidth.

In-memory systems, like the Kove® XPD® [3], provide better latency, endurance and availability as flash chips. Theoretically, the address space of the XPD could be used to deploy a parallel file system, but performance would be limited by the POSIX semantics. The relaxed MPI-IO semantics would enable lock-free access. Since many of the current MPI-IO implementation are optimized for the conventional storage, we believe a in-memory MPI-IO driver for pooled memory deserves a thorough analysis.

The final publication is available at Springer via http://dx.doi.org/10.1007/978-3-319-67630-2_48

Our **contributions** are: 1) we provide an MPI-IO implementation for the pooled memory of the XPD 2) we investigate the performance of the developed MPI-IO driver. While the large and scale-out storage provided by the XPD is valuable by itself, the driver can be considered as an intermediate step towards a burst buffer solution.

This paper is structured as follows: Section 2 discusses related work, then Section 3 and 4 describe the used API and MPI-IO implementation, Section 5 and 6 show the test setup and performance results. Finally, the paper is summarized in Section 7.

2 Related Work

Relevant state-of-the-art can be grouped into performance optimization, burst buffers to speedup I/O and in-memory storage solutions.

Optimization and tuning of file systems and I/O libraries is traditionally an important but daunting task as many configuration knobs can be considered in parallel file system servers, clients and the I/O middleware. Without tuning, typical workloads stay behind the peak-performance by orders of magnitude. With considerable tuning effort a well fitting problem can yield good results: [4] reports 50% peak performance with a single 291 TB file. In [5] MPI-IO and HDF5 were optimized and adapted to each other, improving write bandwidth by 1.4x to 33x.

Many existing workloads can take benefit of a burst buffer as fast write-behind cache that transparently migrates data from the fast storage to traditional parallel file system. Burst buffers rely on flash or NVRAM to support random I/O workloads. For flash based SSDs many vendors offer high-performance storage solutions, for example, DDN Infinite Memory Engine (IME) [6], IBM FlashSystem [7] and Cray's DataWarp accelerator [8]. Using comprehensive strategies to utilize flash chips concurrently, these solutions are powerful and robust to guarantee availability and durability of data for many years.

The integration of Cray DataWarp burst buffer into the NERSC HPC architecture [9] increased the I/O performance of Chumbo-Crunch simulator by 2.84x to 5.73x, compared to Lustre. However, for the sake of efficient burst buffer usage, the serial simulator workflow had to be split into single stages (i.e. simulation, visualization, movie encoding), which then were executed in parallel. The research group at JSC uses DDN IME burst buffer [10] and GPFS to identify requirements for the next HPC generation. The main purpose is to accelerate the I/O performance of the NEST ("NEural Simulation Tool"). The preliminary IOR experiments show, that I/O performance can be increased upto 20x. BurstFS [11] uses local NVRAM of compute nodes, instead of dedicated remote machines. An elaborated communication scheme interconnects the distributed NVRAM and provides a contiguous storage space. This storage is allocated at beginning and exists for the lifetime of the job. In the experiments, BurstFS outperforms OrangeFS and PLFS by several times.

In [12], a user-level InfiniBand-based file system is designed as intermediate layer between compute nodes and parallel file system. With SSDs and FDR Infiniband, they achieve on one server a throughput of 2 GB/s and 3 GB/s for write and read, respectively.

The usage of DRAM for storing intermediate data is not new and ramdrives have been used in MSDOS and Linux (with tmpfs) for decades. However, offered RAM storage was used as temporary local storage and not durable and usually not accessible from remote nodes. Exporting tmpfs storage via parallel file systems has been used mainly for performance evaluation but without durability guarantees. Wickberg and Carothers introduced the RAMDISK Storage Accelerator [13] for HPC applications that by flushes data to a backend. It consists of a set of dedicated nodes that offer in-memory scratch space. Jobs can use the storage to prefetch input data prior job execution or as write-behind cache to speedup I/O. A prototype with a PVFS-based RAMDISK improved performance of 2048 processes compared to GPFS (100 MB/s vs. 36 MB/s for writes). Burst-mem [14] provides a burst buffer with write-behind capabilities by extending Memcached [15]. Experiments show that the ingress performance grows up to 100 GB/s with 128 BurstMem servers. In the field of big data, in-memory data management and processing has become popular with Spark [16]. Now there are many software packages providing storage management and compute engines [17].

The Kove XPD [3] is a robust scale-out pooled memory solution that allows to aggregate multiple Infiniband links and devices into one big virtual address space that can be dynamically partitioned. Internally, the Kove provides persistency by periodically flushing memory with a SATA RAID. Due to the performance differences, the process comes with a delay, but the solution is connected to a UPS to ensure that data becomes durable in case of a power outage. While providing many interfaces, the XPD does not offer a shared storage that can be utilized from multiple nodes concurrently.

3 XPD KDSA API

The XPD KDSA API is a low-level API that allows to send and receive data using write/read calls by utilizing RDMA. Data can be transferred synchronously or asynchronously, additionally, memory can be pre-registered for use with the Infiniband HCA. Since registration of memory is time consuming, for unregistered memory regions, the system may either use an internal (pre-registered) buffer and copy the user's data to the buffer, or for larger accesses it registers the memory, performs an RDMA data transfer and then unregisters the memory.

To address an XPD volume as a virtual address space, the XPD uses a connection specifier in the form: <local_address>/<server>.<link>:<volume ID>. Multiple volumes and client or server links can be aggregated by adding them with a +, data is then striped across these volumes/links. Similar to parallel file systems, this allows to scale the number of connections with the requirements. Upon connecting to an XPD, a client spawns a thread per volume to drive the

I/O, flags can control its behavior. To improve latency, this thread can use spin locks to wait for requests and transfer the data or it conserves CPU time by only becoming active upon events. The latter option is chosen as default for our driver.

4 XPD-MPIIO-Driver

The driver[1] is implemented as a shared library and usable with any MPI. It can be selected at startup of an application using LD_PRELOAD with the shared library. All implemented routines check the file name for the prefix "xpd:". Without the prefix, they route the accesses to the underlying MPI. Thus, files can be selectively stored on XPD volumes.

The file driver implements important functions utilizing the relaxed consistency semantics offered by MPI-IO: MPI_File_open, close, delete, get_position, get_size, preallocate, read_at, write_at, read_at_all, write_at_all, read, write, seek, set_size, set_view and sync. Collective read/write are calling the independent counter part. The selection is inspired by the needs of HDF5 and IOR. Note that the driver does not cache any I/O on the client side.

The implementation comes with a few limitations: Since we do not know the memory regions, the KDSA calls for unregistered memory are used implying overhead as described above. During the open/close the Infiniband connections to the XPD's are established and destroyed. This causes additional overhead but offers the freedom to choose the volumes on a file basis. Partial support for file views as needed by NetCDF4/HDF5 is provided.

Internally, the file driver uses the shared memory space provided by one or multiple XPD volumes. It records the actual file size at the beginning of this memory region but cannot grow beyond the aggregated size of the volumes. Each process tracks its view of the file size and exchanges this information upon file close or flush as needed by MPI-IO semantics. The data space is not initialized with zeros, which is an issue if files are written in a sparse format. Since for many use cases, the file is completely overwrite, this is not a show stopper – for instance, with fill-values, NetCDF/HDF5 initializes the data regions. A formatting tool is contained in the repository that initializes file size (alternatively call MPI_file_delete()) or completely zeroes memory regions.

5 Test Setup

5.1 Testsystems

The tests with the XPD were run on Cooley, the visualization cluster of Mira on ALCF. It provided three XPD's with a total of 14 FDR connections and is connected to a GPFS file system. Each node is equipped with one FDR HCA.

[1] **The code is available as open source.**
`http://github.com/JulianKunkel/XPD-MPIIO-driver`

To investigate the difference between XPD and other state-of-the-art HPC systems, we run some benchmarks on Cooley's GPFS and many on DKRZ's supercomputer Mistral. Mistral hosts 3000 compute nodes each equipped with an FDR interconnect and a Lustre storage system with 54 PByte capacity. The peak transfer rate of the file system we used is 450 GiB/s[2]. When we conducted our measurements, the phase 2 storage system was almost unused by other users.

5.2 Benchmarks

As our primary benchmark, IOR [18] is used varying access granularity, processes-per-node, nodes, XPD connections and access pattern (random and sequential). In all cases MPI-IO with independent I/O is measured. IOR is used with a transfer size equal to the access granularity and 20 GiB of data per XPD connection (and volume)[3]. To synchronize the measurements the inter-phase barriers were turned on (IOR option -g). For the Lustre benchmarks we were trying to reuse the XPD parameters wherever possible. Collective buffer was enabled for write operations smaller than 512 KiB, we configured MPI-IO to use one aggregator per node and, in all cases the number of stripes was twice as much as the number of nodes.

Finally, to measure performance of individual operations to investigate variability, the sequential benchmark `io-modelling` is used[4] It uses a high-precision timer and supports various access patterns on top of the POSIX interface.

6 Evaluation

The goal of our evaluation is to systematically investigate the scaling behavior of the Kove XPD's. The following experiments are conducted: 1) scaling clients for 14 connections; 2) scale-out performance on 14 nodes with increasing number of connections; 3) variability of performance; Additionally, a comparison to DKRZ's Lustre system is made and some results are obtained on Cooley's GPFS system.

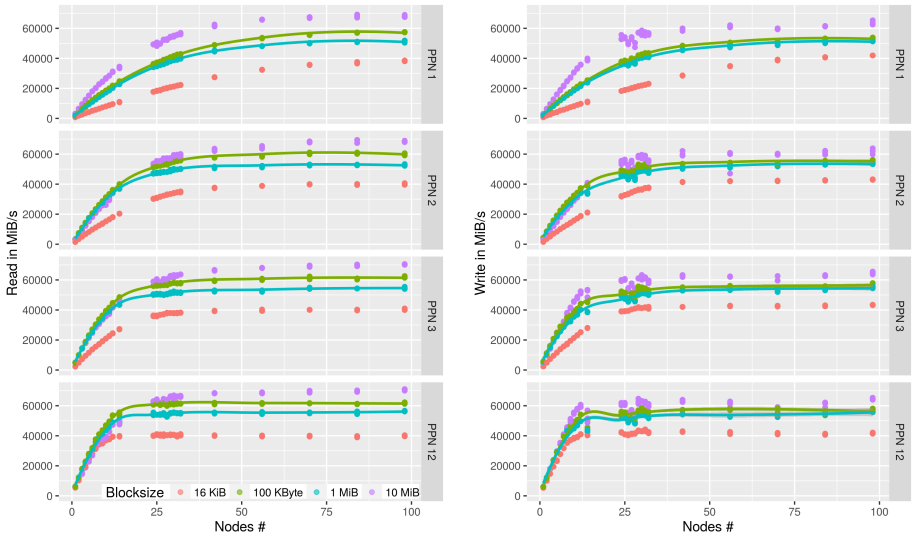
Since the storage capacity is rather small (files up to 100 GiB have been accessed) compared to the speed of the tests, the time for open/close are investigated explicitly in experiment 4). In average across all conducted experiments, the time of open/close reduces the reported performance of the XPD by 10%. However, for production runs, larger files and capacities are assumed, reducing this overhead. Therefore, the performance reported subsequently in this paper is reported without the open/close time.

Note that on the XPD sequential and random I/O behave similarly due to the DRAM storage and, thus, we usually report random performance.

[2] `http://www.vi4io.org/hpsl/2016/de/dkrz/lustre02`

[3] The memory capacity of the XPD's is shared amongst all users, therefore, we had access to 14 volumes each 20 GiB.

[4] `https://github.com/JulianKunkel/io-modelling`



(a) Read (b) Write

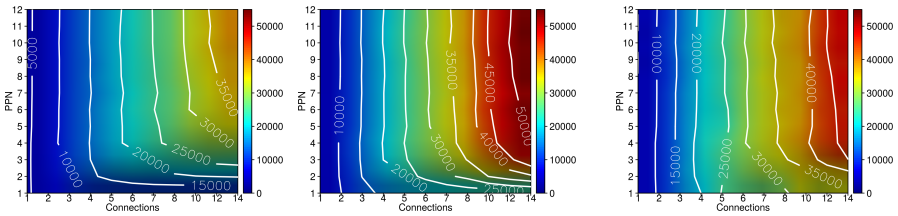
Fig. 1: Performance overview: varying client node count and PPN. The graph contains fitting curves for 100 KiB and 1 MiB blocks.

6.1 Scaling the number of clients

In this first experiment, the maximum number of available volumes and IB links available are used (14).

Figure 1 shows the achieved performance for 1 to 98 client nodes and 1 to 12 processes per node (performance between 3 and 12 PPN is between the measurements). Under optimal conditions, the performance should increase linearly from 1 to 14 nodes as each is equipped with one IB FDR HCA and then it should saturate the network. Assuming roughly 6 GiB/s throughput for the FDR link, 84 GiB/s of performance should be observable.

Observations: 1) read/write behave mostly symmetrically, i.e., good read performance implies good write performance; 2) performance increases nearly with the number of client nodes and then saturates, but with PPN=1 it scales beyond 14 client nodes; 3) for small access granularities, the workload is dominated by the latency of IB and the compute overhead, thus, it improves beyond 14 client nodes and using more PPN; 4) for large access granularities, a high percentage of peak is achieved quickly. Overall, 14 nodes with 12 PPN saturate at least 50% of the available network throughput and 24 clients reach almost peak; 5) performance of 100 KByte accesses is higher than for 1 MiB in many cases. This is due to the pre-registered memory region inside the KDSA library. This buffer is used for small accesses but not for 1 MiB. Therefore, the overhead for memory registration is added which slows down the I/O.



(a) Granularity: 16 KiB (b) Granularity: 100 KByte (c) Granularity: 10 MiB

Fig. 2: Read performance with variable connections and PPN. Isolines for multiples of 5000 MiB/s are shown.

6.2 Scale-out with multiple connections

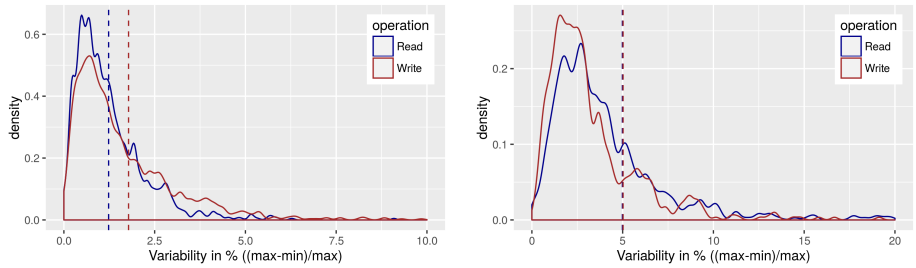
To show the scale-out behavior, the performance when varying PPN and the number of XPD connections has been measured for the fixed configuration of 14 client nodes (that should theoretically be able to saturate all XPD connections). Figure 2 shows a heat-map for different block granularities. This gives us also another perspective to investigate scaling behavior for variable PPN. In the best case, performance increases linearly with the number of connections and is constantly at a high level for variable numbers of PPN.

Observations: 1) for large accesses, the performance isolines show that about 25 GB/s are achievable per connection up to 5 connections regardless of the PPN; 2) starting with 6 connections, multiple PPN are needed to drive I/O and the scaling is not optimally any more. Still, as seen in Figure 1, more PPNs and about 24 client nodes would increase throughput to 60 GiB/s; 3) smaller granularities also yield good performance with PPN=1, but the hill like structure shows that multiple PPNs are necessary to drive the latency bound I/O. Overall, the system scale well when increasing the number of XPD connections and servers.

6.3 Performance Variability

The variability of access time has been investigated. When re-running an experiment, the overall performance of the repeated run should exhibit a similar performance behavior. Since each experiment takes at least several seconds to complete, we additionally investigated the runtime of repeatedly invoking the same I/O call.

A comparison of the runtime of the three repeats for each individual configuration ($\frac{min-max}{max}$) reveals the variability when re-running an experiment. On the XPD, the arithmetic mean value of variability is 1.23% for read and 1.78% for write accesses, albeit the mean runtime of an experiment was only about 10s. Thus, on average, when repeating an experiment, performance can be 1.8% worse than in the best case. Across the experiments, Lustre varies about 5% for read and write although its runtime is longer and, thus, less variation is to be



(a) XPD (b) Lustre

Fig. 3: Density of the variability range across all conducted experiments (span across three repeats each).

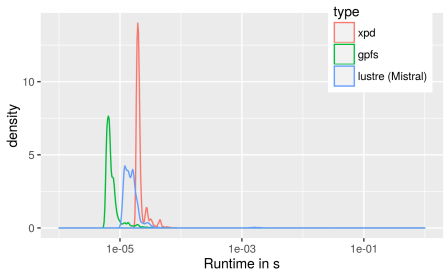
Size	Type	Read			Write		
		XPD	GPFS	Lustre	XPD	GPFS	Lustre
16 KiB	seq	707.8	659.8	522.0	709.8	533.0	778.0
100k	seq	1653.8	1139.2	1082.2	1773.3	611.7	927.7
1 MiB	seq	1837.3	1062.5	996.2	1768.2	629.8	965.9
10 MiB	seq	3401.7	928.3	994.3	3274.3	742.3	916.9
16 KiB	rnd	676.8	1.2	1.5	600.4	71.7	20.6
100k	rnd	1538.5	4.7	9.2	1636.1	346.7	80.6
1 MiB	rnd	2052.6	29.6	49.2	1967.1	184.6	157.6
10 MiB	rnd	3456.6	301.2	277.6	3335.6	430.0	352.1

Table 1: Variability test: mean performance in MiB/s over the runtime

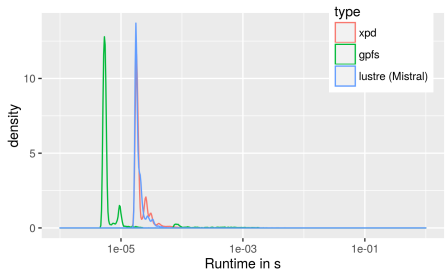
expected. The density (similar to a fine-grained histogram) for all experiments is shown in Figure 3.

Performance variability with individual I/Os. This experiment is conducted measuring timing of 10,000 individual I/Os with a single process on Cooley's XPD and GPFS, and on DKRZ's Lustre. The density plots of measuring these results is shown in Figure 4. This graph shows the qualitative difference between the file systems. The mean performance for each experiment is shown in Table 1, i.e., the average performance when timing a complete run; naturally, a few very slow operations lead to a significant reduced mean performance.

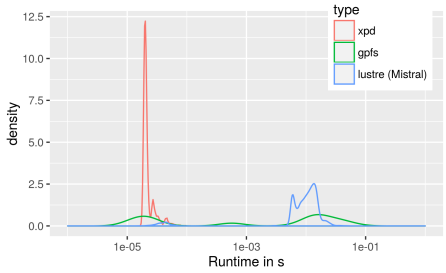
Observations: As suggested by comparing application runs, the XPD's performance does not vary much between individual I/Os, i.e., the observed runtime always forms a group. While some reads in the optimized sequential I/O can perform as fast as on the XPD – i.e., with wire speed, most operations do not and, obviously, random I/O from parallel file systems is significantly slower. Actually, for sequential reads, in combination with caching, the read-ahead and write-behind strategy of Lustre and GPFS can result in faster performance than the



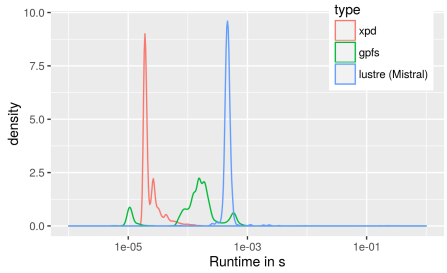
(a) 16 KiB sequential read



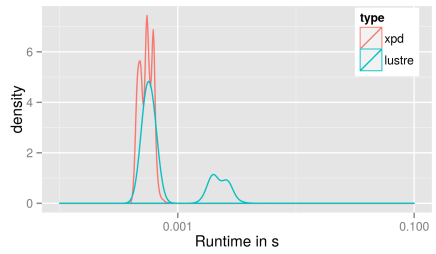
(b) 16 KiB sequential write



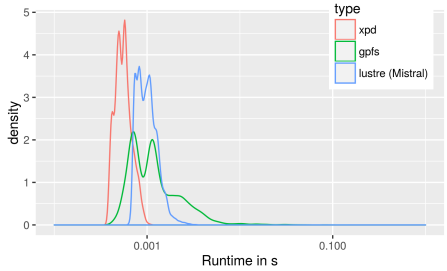
(c) 16 KiB random read



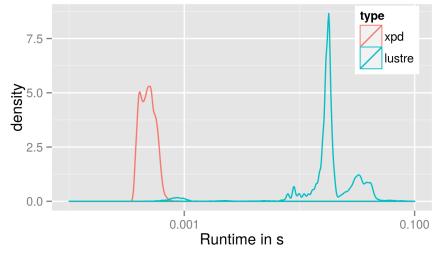
(d) 16 KiB random write



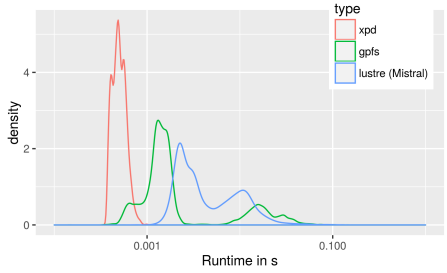
(e) 1 MiB sequential read



(f) 1 MiB sequential write



(g) 1 MiB random read



(h) 1 MiB random write

Fig. 4: Density of timing individual I/O operations

XPD for individual operations. Still, the mean performance over the complete experiment, i.e., when doing all 10,000 operations is faster in all cases except for sequential write of 16 KiB of data on Lustre. The reason is the reduced performance variability on the XPD.

6.4 Additional Experiments

Besides, we investigated open/close behavior of XPDs, and could create a linear model for prediction of open/close times. For example, our model says that for NN=500 nodes and PPN=12 an open time will be about 2.5 seconds. On Lustre using the same parameters we observed open times around 0.7 seconds.

Furthermore, we measured performance of NetCDF4 on XPDs, GPFS and Lustre using collective, independent, and chunked I/O modes. Our main observation in these experiments was that XPDs are insensitive to the different I/O modes, whereas in experiments with GPFS and Lustre we could see an irregular I/O behavior.

7 Summary

Storage on XPDs significantly outperforms our Lustre system in the small-blocks random I/O benchmarks. In this case and in contrast to XPD, the increasing number of nodes and processes accessing the storage don't provided the desired scaling effect. The performance benefit of the XPD is smaller when we use large access granularities. While we have not exploited all available tuning knobs for Lustre and GPFS, it becomes apparent that the MPI-IO driver on top of the XPD outperforms GPFS and Lustre. Also, with our MPI-IO driver, the need to tune too many knobs vanishes, users can rely on the performance without changing one of many parameters as needed for other file systems. For application relevant workloads using NetCDF, XPD is relatively insensitive to various settings of the I/O method and chunking. It simply scales with the number of processes and nodes up to a rather predictable throughput of 4 GiB/s per client node. In particularly, due to the nature of the storage technology, the I/O variance is much less than for other file systems leading to much better performance predictability. From these results, it appears that this MPI-IO driver supports I/O heavy workloads. A burst buffer system equipped with a set of XPDs has potential for improvement of I/O performance by several times.

Acknowledgment

Thanks to Kove for their support and discussion. Thanks to our sponsor William E. Allcock for providing access and feedback. This research used resources of the Argonne Leadership Computing Facility, which is a DOE Office of Science User Facility supported under Contract DE-AC02-06CH11357.

References

1. Liu, N., Cope, J., Carns, P., Carothers, C., Ross, R., Grider, G., Crume, A., Maltzahn, C.: On the role of burst buffers in leadership-class storage systems. In: In Proceedings of the 2012 IEEE Conference on Massive Data Storage. (2012)
2. Romanus, M., Parashar, M., Ross, R.B.: Challenges and considerations for utilizing burst buffers in high-performance computing. arXiv preprint 1509.05492 (2015)
3. Kove Corporation: about xpress disk (xpd). (2015)
4. The HDF Group: A Brief Introduction to Parallel HDF5. `https://www.alcf.anl.gov/files/Parallel_HDF5_1.pdf`
5. Howison, M., Koziol, Q., Knaak, D., Mainzer, J., Shalf, J.: Tuning HDF5 for Lustre File Systems. In: Workshop on Interfaces and Abstractions for Scientific Data Storage (IASDS10)„ Heraklion, Crete, Greece, September 24, 2010. (2012)
6. DDN: Worlds's most advanced application aware I/O acceleration solutions. `http://www.ddn.com/products/infinite-memory-engine-ime14k`
7. IBM: Flash Storage. `http://www-03.ibm.com/systems/storage/flash`
8. Cray: CRAY XC40 DataWarp Applications I/O Accelerator. `http://www.cray.com/sites/default/files/resources/CrayXC40-DataWarp.pdf`
9. Ovsyannikov, A., Romanus, M., Straalen, B.V., Weber, G.H., Trebotich, D.: Scientific workflows at datawarp-speed: Accelerated data-intensive science using nersc's burst buffer (2016)
10. Schenck, W., El Sayed, S., Foszczynski, M., Homberg, W., Pleiter, D. In: Early Evaluation of the "Infinite Memory Engine" Burst Buffer Solution. Springer International Publishing, Cham (2016) 604–615
11. Wang, T., Mohror, K., Moody, A., Sato, K., Yu, W.: An ephemeral burst-buffer file system for scientific applications. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. SC '16, Piscataway, NJ, USA, IEEE Press (2016) 69:1–69:12
12. Sato, K., Mohror, K., Moody, A., Gamblin, T., de Supinski, B.R., Maruyama, N., Matsuoka, S.: A user-level infiniband-based file system and checkpoint strategy for burst buffers. In: Cluster, Cloud and Grid Computing (CCGrid), 2014 14th IEEE/ACM International Symposium on, IEEE (2014) 21–30
13. Wickberg, T., Carothers, C.: The RAMDISK storage accelerator: a method of accelerating I/O performance on HPC systems using RAMDISKs. In: Proceedings of the 2nd International Workshop on Runtime and Operating Systems for Supercomputers, ACM (2012) 5
14. Wang, T., Oral, S., Wang, Y., Settlemyer, B., Atchley, S., Yu, W.: BurstMem: A high-performance burst buffer system for scientific applications. In: Big Data (Big Data), 2014 IEEE International Conference on, IEEE (2014) 71–79
15. Jose, J., Subramoni, H., Luo, M., Zhang, M., Huang, J., Wasi-ur Rahman, M., Islam, N.S., Ouyang, X., Wang, H., Sur, S., et al.: Memcached design on high performance RDMA capable interconnects. In: 2011 International Conference on Parallel Processing, IEEE (2011) 743–752
16. Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauley, M., Franklin, M.J., Shenker, S., Stoica, I.: Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In: Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation, USENIX Association (2012) 2–2
17. Zhang, H., Chen, G., Ooi, B.C., Tan, K.L., Zhang, M.: In-memory big data management and processing: A survey. IEEE Transactions on Knowledge and Data Engineering **27**(7) (2015) 1920–1948

18. Loewe, W., McLarty, T., Morrone, C.: IOR Benchmark (2012)