Performance Evaluation and Modeling of HPC I/O on Non-Volatile Memory

Wei Liu wliu34@ucmerced.edu
Kai Wu kwu42@ucmerced.edu
Jialin Liu* jalnliu@lbl.gov
Feng Chen† fchen@csc.lsu.edu
Dong Li dli35@ucmerced.edu
University of California, Merced *Lawrence Berkeley National Lab †Louisiana State University

***Abstract*—HPC applications pose high demands on I/O performance and storage capability. The emerging non-volatile memory (NVM) techniques offer low-latency, high bandwidth, and persistence for HPC applications. However, the existing I/O stack are designed and optimized based on an assumption of disk-based storage. To effectively use NVM, we must re-examine the existing high performance computing (HPC) I/O subsystem to properly integrate NVM into it. Using NVM as a fast storage, the previous assumption on the inferior performance of storage (e.g., hard drive) is not valid any more. The performance problem caused by slow storage may be mitigated; the existing mechanisms to narrow the performance gap between storage and CPU may be unnecessary and result in large overhead. Thus fully understanding the impact of introducing NVM into the HPC software stack demands a thorough performance study.**

In this paper, we analyze and model the performance of I/O intensive HPC applications with NVM as a block device. We study the performance from three perspectives: (1) the impact of NVM on the performance of traditional page cache; (2) a performance comparison between MPI individual I/O and POSIX I/O; and (3) the impact of NVM on the performance of collective I/O. We reveal the diminishing effects of page cache, minor performance difference between MPI individual I/O and POSIX I/O, and performance disadvantage of collective I/O on NVM due to unnecessary data shuffling. We also model the performance of MPI collective I/O and study the complex interaction between data shuffling, storage performance, and I/O access patterns.

I. Introduction

Modern high performance computing (HPC) applications are often characterized with huge data sizes and intensive data processing. For example, the Blue Brain project aims to simulate the human brain with a daunting 100PB memory that needs to be revisited by the solver at every time step; the cosmology simulation studying Q continuum works on 2PB per simulation. Both of these simulations require transformation of data representation, which poses high demands on I/O performance and storage capability.

The emerging Non-volatile Memory (NVM) techniques, such as Phase Change Memory [1] and STT-RAM [2], offer low-latency access, high bandwidth, and persistency. Their performance is much better than the traditional hard drives, and close to or even match that of DRAM. The non-volatility and high performance of NVM blur the line between storage and main memory, hinting at opportunities to overhaul classical IO system and memory hierarchies. Table I summarizes the characteristics of different NVM technologies and compares them to traditional DRAM and storage technologies.

TABLE I: Memory Technology Summary [3]

	Read time (ns)	Write time (ns)	Read BW (MB/s)	Write BW (MB/s)
DRAM	10	10	1,000	900
PCRAM	20-200	80-10^4	200-800	100-800
SLC Flash	10^4-10^5	10^4-10^7	0.1	10^{-3}-10^{-1}
ReRAM	5-10^5	5-10^8	1-1000	0.1-1000
Hard drive	10^6	10^6	50-120	50-120

The emergence of NVM has compound impacts on the existing HPC systems and applications. Given the high performance and non-volatility of NVM, we must re-examine the existing I/O system to properly integrate NVM into it. Using NVM as a fast storage, the previous assumption on the inferior performance of storage, such as disk drives, is not valid any more. The performance problem caused by slow storage may be mitigated; The performance bottleneck along the I/O path may be shifted from storage to other middle-level system components; The existing mechanisms to narrow the performance gap between storage and CPU may be unnecessary and result in undesirable overhead.

In this paper, we analyze the performance of I/O intensive HPC applications with NVM as the high-speed block device. Given its high compatibility, we anticipate that such a block-based NVM model is likely to become the mainstream in industry (e.g., the recently announced Intel Optane [4]) and be adopted in the near future soon . We pose the following questions to gain important insight into the application performance with NVM.

- What is the impact of NVM on the performance of traditional page cache? Is it still reasonable to use page cache for NVM-based storage?
- Comparing MPI individual I/O and POSIX I/O based on NVM, what is their performance difference in the HPC domain? With a high-speed NVM device, would MPI individual bring too much overhead because it brings one extra layer on top of POSIX I/O?
- MPI I/O introduces collective I/O techniques to optimize application performance, based on the assumption of poor I/O performance. Is it still valid to use those techniques under the deployment of NVM?

To answer the above questions, we use a set of representative HPC applications to evaluate their performance based

on Intel's Persistent Memory Block Driver (PMBD) [5]. We make several findings through our study.

- The benefits of page cache is diminished with the deployment of NVM, but still plays an important role to improve I/O performance. Comparing with SSD and regular hard drive, NVM is less sensitive to page cache size when the working set size of the application is very large. This is due to the superior performance of NVM. However, when the working set can be accommodated in page cache, NVM does not exhibit significant performance advantages over SSD and hard drive.
- MPI individual I/O and POSIX I/O have minor performance difference with the existence of NVM. The overhead of MPI individual I/O is not pronounced, even if we use NVM as a fast storage. In a single-node deployment, MPI individual I/O performs only 4.87% worse than POSIX I/O. In a multiple-node deployment, there is almost no performance difference between the two. This indicates that given the current highly optimized implementation of MPI individual I/O, the performance overhead of MPI individual I/O would not become a problem for the future HPC, even if we have a fast storage device, such as NVM.
- MPI collective I/O can perform worse than MPI individual I/O with the deployment of NVM. MPI collective I/O aims to aggregate I/O operations to improve performance of MPI individual I/O. However, the data shuffling cost in MPI collective I/O is often larger than the performance benefit of collective I/O, given the high speed of NVM. For example, our results show that using collective I/O for a workload with random I/O data accesses from multiple MPI processes performs 38.4% worse than using MPI individual I/O for the same workload in NVM.

Based on our observations, in this paper we further introduce a performance model to analyze the tradeoff between I/O aggregation overhead and benefit. Based on the model, we explore how the collective I/O should be employed with the upcoming NVM technology.

The paper proceeds as follows. Section II covers the background. Section III presents application performance on NVM under various test environments. Section IV introduces our performance model for the MPI collective I/O. We discuss related work and conclude in Sections V and VI, respectively.

II. Background

A. NVM Usage Model

Drawing a blurry line between traditional volatile memory and persistent storage, NVM has at least two usage models.

(1) **Memory-based Model**. NVM is treated as the regular, byte-addressable main memory: NVM is attached to the memory bus in form of DIMMs and directly managed by the memory controller. The NVM space is exposed to the host as part of physical memory address space, which could be directly accessed through `load` and `store` instructions. To bridge the potential performance gap between NVM-only main memory and the traditional DRAM-only main memory, NVM could be paired with a small portion of DRAM to mitigate intensive writes and enhance lifetime. On one hand, such a memory-based model provides high performance and directly opens many attractive properties, such as byte addressability and persistence, to applications. On the other hand, this model introduces high complexity to programmers, especially for handling data integrity and consistency issues upon power and system failure. Prior studies, such as Mnemosyne [6], CDDCS [7], and NV-heap [8], aim to provide an easy and flexible programming interface to alleviate such a programming burden. Also, in order to fully exploit the potential of memory-based model, applications have to be redesigned to fit this model, which introduces backward compatibility issues.

(2) **Storage-based model**. Another model is to use NVM as a block device, similar to traditional HDD or SSD: NVM can be used to directly displace NAND flash in an SSD and managed by an I/O controller. The host can access the device through a regular block I/O interface (e.g., PCI-E or SATA) via `read` and `write` commands. Limited by the I/O bus bandwidth, the storage-based model cannot fully exploit its potential, such as byte-addressability. However, this scheme provides a maximum compatibility to the existing applications and operating systems, which allows it to be a simple drop-in solution. A user can simply use an NVM device as a regular flash SSD, create partition and file systems atop, and immediately enjoy the high I/O speed. Recently Intel announced their 3D XPoint based product, called Optane, which is a PCI-E device based on the block device model [4]. In this work, We assume a storage-based model in this work, which is the most practical NVM solution in the near future.

B. MPI Collective I/O

In conventional disk based storage, I/O performance is highly sensitive to not only the amount of data being accessed but also the access pattern (e.g., sequential vs. random). In an MPI-based application, multiple I/O streams could be issued individually and independently from multiple MPI processes, which is normally considered as the worst situation for disk drives, because this situation creates a disk head's "seek storm" and causes performance loss. Thus, creating a disk-friendly access pattern is an important consideration by MPI I/O.

Collective I/O is a mechanism to improve MPI-based parallel I/O performance. The basic idea of MPI collective I/O is to scatter and gather data between MPI processes that need to perform I/O operations. Such scatter and gather operations are performed by only a limited number of MPI processes, named as *aggregator*. Each aggregator coalesces I/O requests and iteratively performs I/O operations for all MPI processes or a subset of them. Figure 1 depicts the MPI collective I/O scheme for write operation. Read operation happens similarly but in an opposite data path. In the figure, there are two aggregators (MPI processes 1 and 2). Each aggregator gathers data from all MPI processes in two iterations. Then each aggregator coalesces the data and writes into persistent storage.

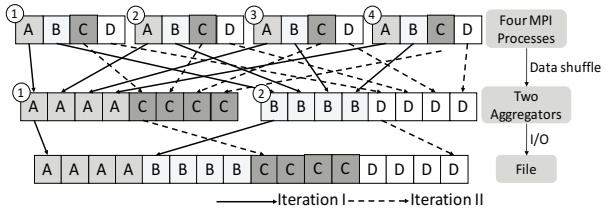

Fig. 1: The MPI collective I/O scheme. The numbers in circles are MPI process IDs. There are two aggregators (MPI processes 1 and 2) in this example. Letters A, B, C, and D represent data from four contiguous blocks on NVM.

The collective I/O approach reduces the number of I/O transactions, enables contiguous I/O operations, and avoids fetching useless data, effectively improving I/O performance for certain workloads. However, MPI collective I/O also brings the so-called "data shuffling" overhead, which is associated with the process of data gathering (for write operations) and scattering (for read operations).

Given the poor performance of conventional storage devices, the data shuffling overhead is often overweighted by performance benefits of optimized I/O operations from MPI collective I/O. However, with high-speed solid state storage, such as NVM and SSD, which are relatively insensitive to I/O patterns (e.g., random accesses) and deliver much higher I/O performance, MPI collective I/O may not always remain advantageous.

The current MPI library also allows individual I/O, where MPI processes conduct I/O operations individually without the coordination of MPI collective I/O and do not involve data shuffling.

C. Benchmarks

For our experimental study, we have carefully selected four representative I/O intensive HPC benchmarks.

1) MADBench2: This benchmark is a "stripped-down" version of MADCAP (a Microwave Anisotropy Dataset Computational Analysis Package) [9]. MADBench2 has an I/O mode that performs MPI I/O in three phases, S, W, and C. The three phases have complicated write-only, read-only, and read/write operations respectively.

2) IOR: IOR is a benchmark widely used to study parallel I/O performance at both POSIX and MPI-IO levels [10]. It is highly configurable and supports various I/O patterns, including "sequential" and "random offset" file access, and individual I/O and collective I/O.

Several IOR configuration parameters are related to our work, including "segment count", "block size", and "transfer size", shown in Figure 2. For collective I/O, the given data in an MPI process is partitioned into segments, and then each segment is further partitioned into blocks. During the data shuffling phase, an MPI process in each iteration of the data shuffling sends or receives at most "transfer size" of data. IOR also has a parameter, called "reorder tasks to random", which enables random I/O accesses. We use this option for IOR throughout the paper.

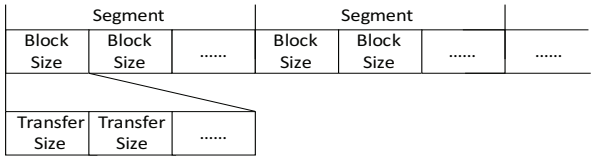

Fig. 2: Configuration parameters for IOR benchmark.

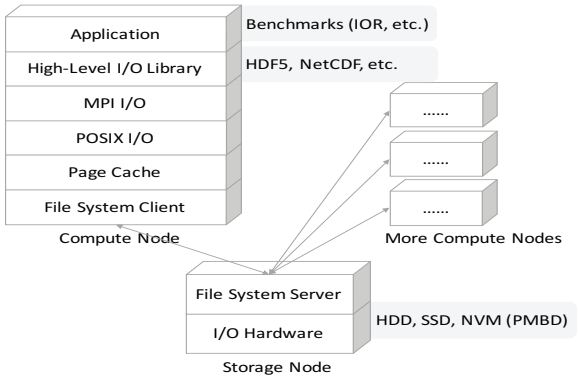

Fig. 3: HPC I/O system Hierarchy

3) HACC-IO: This benchmark is the I/O kernel of HACC (an HPC application based on N-body simulation) [11]. It has random I/O write operations with all-to-all communication patterns. This benchmark allows us to configure the number of particles ("numparticles") simulated in HACC-IO to change the workload size. The total amount of data to write is the "numparticles" multiplied by the number of MPI processes.

4) S3aSim.: This benchmark is an MPI-IO based sequence similarity search algorithm framework [12]. S3aSim emulates IO access patterns in mpiBLAST [13], which is "streaming-like", read-only data accesses. S3aSim has five working phases, and we focus on one of the phases (i.e., the I/O phase).

D. PMBD Emulator

As NVM devices are not available in the market, we use Persistent Memory Block Driver (PMBD) [14], which is a DRAM based NVM emulator driver, for our experiments. PMBD is a light-weight PM (Persistent Memory) block driver based on an OS kernel module in Linux 2.6.34. It reserves a portion of DRAM-based physical memory space by changing the e820 table in the high memory address space. PMBD provides a standard block I/O interface after being loaded into OS as a regular block device, on top of which partitions and file systems can be created. Internally, the PMBD driver is responsible for mapping the logical block addresses to physical memory pages, receiving the incoming `read` and `write` commands, and translating them to `load` and `store` instructions. From the perspective of application level software and other system components, a PMBD device has no difference from other physical block devices, while it provides configurable features of NVM devices, such as emulating various bandwidths, latencies, protections, etc.

E. HPC I/O Hierarchy

The I/O stack in a typical HPC system has multiple layers, shown in Figure 3. The block devices at the bottom level

provide data persistence. Given the variety of different storage devices (e.g., HDD, SSD, PMBD), raw data access latencies range from microseconds to milliseconds, and are sensitive to distinct access patterns (e.g., sequential vs. random). To alleviate the impact of slow I/O operations, the page cache layer in the operating system attempts to hold the workload's working set in memory, satisfying most data accesses in DRAM. Due to its "filtering" effect, the page cache can have a strong impact on I/O performance. The file system layer is responsible for managing storage devices and provides a file system abstraction to allow applications to access storage devices, either connected locally or remotely. In our study, we use network file system (NFS) for remote storage access. MPI I/O built on top of POSIX I/O enables coordinated and remote I/O accesses for MPI processes.

III. Performance Study

We present our performance analysis results in this section. We deploy our tests in a local cluster. Each node of the cluster has two Intel Xeon E5-2630 processors (2.4GHz) with 32GB DDR memory. All nodes in the cluster are connected through 1Gb Ethernet interconnect. We use three types of block devices: one is a regular hard drive (Seagate Constellation.2 500GB hard drive attached by SATA, notated as "HDD" in this section), one is an SSD (Intel SSD730 240GB attached by SATA, notated as "SSD" in this section), and the third is an NVM device emulated with PMBD. NVM is configured with the same bandwidth and latency as DRAM. We use MPICH-3.2 for MPI throughout the paper.

A. Impact of Page Cache

The page cache is a transparent cache for pages originating from a secondary storage device. The operating system (OS) keeps a page cache, which enables quicker accesses to those frequently accessed pages and improves performance. We measure the performance of the three I/O devices with different page cache configurations and study the impact of the page cache on the observed application performance.

We use three benchmarks in our tests, HACC-IO, MADBench2, and S3aSim. The benchmarks are compiled with gcc 4.4.7 and Open MPI-1.10.0. We use one node with four MPI processes for our tests. Figures 4, 5, and 6 show the results for HACC-IO, MADBench2, and S3aSim, respectively.

HACC-IO in Figure 4 simulates 13,107,200 particles in total (i.e., numparticles=13,107,200). It computes and then generates about 2GB data for four MPI processes, and writes them into the three block devices. The figure reveals that the page cache plays an important role to improve performance for HDD and SSD, while it has a limited impact on the performance of NVM. When the page cache size is large (e.g., 9GB and 11GB), there is almost no performance difference between the three devices, because most of the I/O data is cached in the page cache. However, as we reduce the page cache size, there is significant performance difference between the three devices. In general, decreasing the cache size from 11GB to 1GB, the performance of this workload on HDD

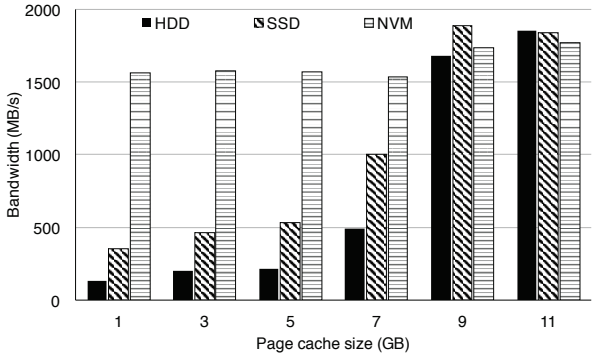

Fig. 4: The performance study for the impacts of page cache on HACC-IO.

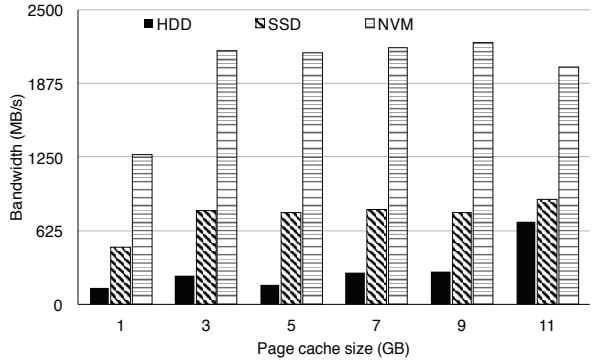

Fig. 5: The performance study for the impacts of page cache on MADBench2.

and SSD is reduced by 92.7% and 84.8% respectively, while the performance with NVM is only reduced by 11.5%. This example illustrates well that with high-speed NVM, the effect of the page cache is weakened.

MADBench2 in Figure 5 uses a working set size of about 4GB (particularly the parameters NO_PIX, NO_BIN, NO_GANG, and BLOCKSIZE of MADBench2 are set as 5000, 8, 1, and 1024 respectively), larger than that of HACC-IO. The figure presents the performance of the phase W, which includes both read and write operations. MADBench2 tells us a story slightly different from HACC-IO. As we decrease the cache size from 11GB to 3GB, the performance on the three devices remains stable. This is because of the fact that MADBench2 has a larger working set size and the page cache is unable to effectively cache all data, including those for MADBench2 and system. However, NVM performs the best among the three cases due to its high bandwidth.

S3aSim in Figure 6 uses a working set size of 2GB (with 100 total query number, max size of each query as 5,000, and max count of each query as 10,000). Comparing the performance of MADBench2 and S3aSim, we find that they have the same performance trend: the NVM has the best performance in all cases. But when the page cache is reduced from 3GB to 1GB, MADBench2 on NVM has significant performance reduction, 40.16%, while S3aSim on NVM has only 5.91% performance reduction. We attribute such difference in the performance reduction to the distinct data access patterns of the two applications: S3aSim has streaming-like access pattern, hence the page cache cannot work well, no matter how large the page cache size is; for MADBench2, the page cache takes effect, although the caching effect of page cache

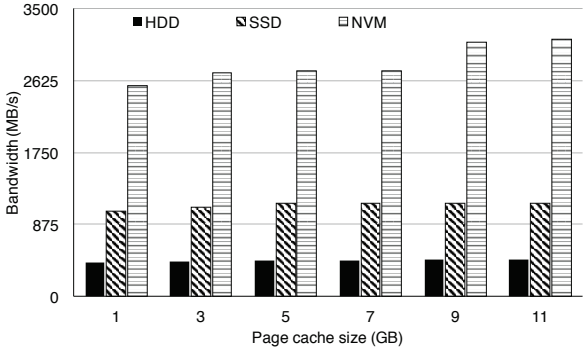

Fig. 6: The performance study for the impacts of page cache on S3aSim.

becomes weaker, when the page cache size is small (1GB).

Conclusions. With the emergence of NVM, the impact of page cache on application performance is diminishing. Compared with the traditional HDD and SSD, NVM is relatively insensitive to the page cache size.

Our study has an important implication on how much page cache space should be allocated for future NVM-based HPC systems. In general, NVM makes it possible to use a smaller page cache, which would save cost and incur ignorable performance impact. We could even explore the possibility of completely bypassing page cache for certain workloads on NVM-based block device, which will save the limited page cache space for other system data and in turn improve the performance of the whole system.

B. POSIX I/O and MPI Individual I/O

MPI I/O is built on top of POSIX I/O (see Figure 3), and is designed to improve the performance of POSIX I/O in the setting of parallel I/O and provide user-friendly I/O abstract. In the system stack, MPI I/O layer ensures data validness for MPI I/O operations and re-organizes data distribution for better performance. However, as an additional layer in the system stack, MPI I/O could introduce certain overhead. With conventional disk storage devices, such overhead is negligible compared to its advantages, however, it could be more pronounced with NVM, because NVM alleviates performance bottleneck at I/O devices and makes the overhead in the other system components more obvious. In this section, we study the performance of MPI individual I/O, and further study the performance of MPI collective I/O in the next section.

We first study the performance of POSIX I/O and MPI individual I/O without the involvement of network communication. In particular, we run the IOR benchmark on a single node. We use 4 MPI processes, each of which performs I/O operations. For the IOR benchmark, we set "block size" as 256MB, "segment count" as 2, and "transfer size" as 16MB, and enable "reorder tasks to random". The final aggregated result file from IOR is a 16GB file (each MPI process writes 4GB data). Figure 7 shows the results.

The figure reveals that there is almost no performance difference between MPI individual I/O and POSXI I/O on a single node for HDD and SSD. However, when we use NVM, we notice that POSIX I/O performs slightly better than MPI individual I/O by 4.87%. We attribute the appearance of such

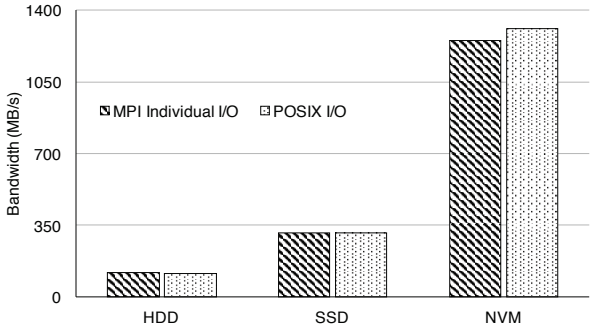

Fig. 7: Comparing the performance of MPI individual I/O and POSIX I/O on a single node with IOR.

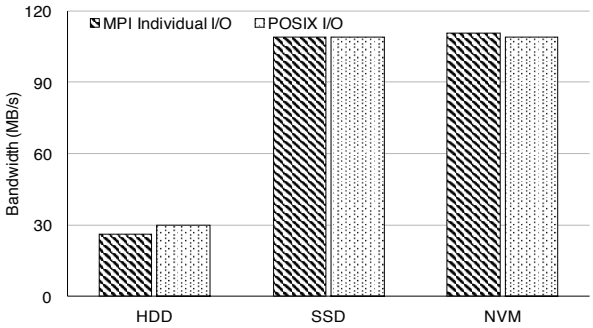

Fig. 8: Comparing the performance of MPI individual I/O and POSIX I/O on multiple nodes with IOR.

performance difference to the better performance of NVM which makes the overhead of MPI I/O more pronounced.

To further study the performance of MPI individual I/O and POSIX I/O, we use five nodes and re-do the tests. Among the five nodes, four nodes run the IOR benchmark with 4 processes per node (16 processes in total), and the fifth node works as a storage node where the other four nodes remotely perform I/O operations. Hence, different from Figure 7, such a deployment has the involvement of communication between the four nodes and the storage node. POSIX I/Os are performed with NFS in our test environment. Figure 8 shows the results.

The figure reveals that MPI individual I/O has almost no performance difference than POSIX I/O in all cases, no matter whether we use HDD, SSD, and NVM. The communication cost in our tests is the major performance bottleneck, much larger than those caused by MPI individual I/O overhead. Hence, the overhead for MPI individual I/O is not clearly spotted in the figure, even if we use a fast storage device, such as SSD and NVM.

Conclusions. The emergence of NVM brings better performance, and also may make some overhead more pronounced than before. In this section, we study the overhead of MPI individual I/O. We find such overhead only sightly impacts performance in a deployment of a single node, and in a multi-node environment, MPI individual I/O has ignorable performance overhead, even if we use NVM. This finding implies that the current implementation of MPI individual I/O is quite efficient, which would introduce little overhead for the future HPC system equipped with NVM.

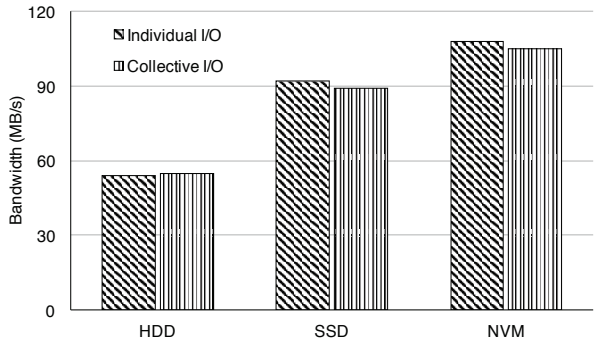

Fig. 9: Comparing the performance of MPI collective I/O and MPI individual I/O (1 process per node) with IOR.

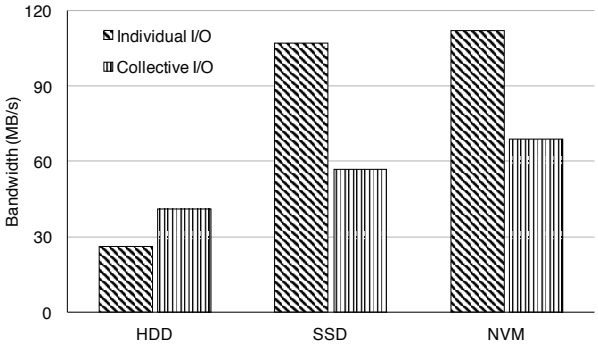

Fig. 10: Comparing the performance of MPI collective I/O and MPI individual I/O (4 processes per node) with IOR.

C. MPI Collective I/O and MPI Individual I/O

MPI collective I/O can bring performance benefit over MPI individual I/O, when I/O operations from MPI processes are interleaved and scattered. By coalescing I/O operations and reorganizing data between MPI processes, MPI collective I/O can reduce the number of I/O transactions and avoid fetching useless data. However, this happens at the cost of data shuffling operations between MPI processes, as discussed in Section II-B. The design of MPI collective I/O is based on a fundamental assumption that the I/O block device is slow and pattern sensitive, such that the data shuffling cost can be overweighted by the performance benefit of using MPI collective I/O. In this section, we study the performance of collective I/O with NVM, and compare the performance of MPI collective I/O and MPI individual I/O.

We use the IOR benchmark and the same configuration (including workload size, block size, and data transfer size) as that for MPI individual I/O and POSIX I/O (Section III-B). We use five nodes for the tests, four of which run IOR, and the fifth node works as a remote storage node for parallel I/O operations. For MPI collective I/O, we use one aggregator per node. Figures 9 and 10 show results for the case of 1 process per node (4 processes in total) and 4 processes per node (16 processes in total), respectively.

The figures reveal that MPI collective I/O brings little benefit in most of cases. For HDD with intensive I/O operations (i.e., 4 processes per node), the collective I/O performs better. But, for SSD and NVM, MPI collective I/O always performs worse than MPI individual I/O.

With conventional HDD, MPI collective I/O demonstrates its performance benefits, even if there is data shuffling cost. However, with the introduction of faster storage device (e.g., SSD and NVM), the I/O cost on the storage device is alleviated, and relatively, the data shuffling cost becomes more pronounced in the overall I/O cost. The results suggest that using MPI individual I/O instead of collective I/O makes more sense for fast storage device due to its low overhead.

Furthermore, we notice that the performance difference between MPI collective I/O and individual I/O becomes bigger in the case of 4 processes per node than in the case of 1 process per node. Such larger performance difference is due to the higher data shuffling cost when dealing with a large number of concurrently running processes.

Conclusions. MPI I/O used to assume slow and pattern-sensitive HDDs as the secondary storage, which makes collective I/O a desirable optimization choice, disregarding the associated small overhead. As storage device performance improves to a point that the performance benefit cannot offset such overhead, MPI collective I/O becomes a detrimental "optimization", especially for NVM. This urges us to revisit other existing mechanisms, besides MPI collective I/Os, that aim to optimize performance based on the ill assumption of slow storage devices. With the emergence of NVM, the existing mechanisms may not be necessary and could be even harmful. In this case, we demonstrate that MPI collective I/O is one of such mechanisms.

In Section IV, we further study the performance of MPI collective I/O and investigate why it has worse performance. We also introduce a performance model that facilitates to make a decision on when to use MPI collective I/O.

IV. Detailed Performance Study for MPI Collective I/O

MPI collective I/O is more than just I/O operations. It includes a set of communication between participating MPI processes before or after I/O operations. We conduct a detailed analysis on the performance of MPI collective I/O.

A. Workflow of MPI Collective I/O

MPI collective I/O performs differently for read and write I/O operations. For read operations, the aggregator processes fetch data from the remote storage node and then redistribute the data among other MPI processes. For write operations, the aggregator processes collect data from other MPI processes and then write the data to the storage node. As discussed in Section II, the whole dataset is partitioned into many data blocks, and the aggregators scatter/gather data among MPI processes iteratively.

Listing 1 shows the workflow for write operations in MPI collective I/O, based on the implementation of MPI collective I/O in MPICH (in particular, ROMIO [15]). In each iteration of MPI collective I/O ($ntimes$ iterations in total), before each collective data write (Line 10), data shuffling is called to gather data from MPI processes (Line 7).

Listing 2 shows the logic of data shuffling in each iteration of MPI collective I/O. Data shuffling is implemented based on MPI asynchronized point-to-point communication (MPI_Irecv/MPI_Isend and MPI_Waitall).

Based on the above discussion, we conclude that MPI collective I/O alternates between data shuffling and I/O operation. In each iteration, data shuffling must be finished before the aggregator starts to write (or read) data. From the view of an individual aggregator, data shuffling and I/O can be treated as blocking operations.

```
1  ADIOI_Exch_and_write(...)
2  {
3    ...
4    for (m=0; m<ntimes; m++) {
5      ...
6      // Shuffling data between MPI processes
7      ADIOI_R_Exchange_data(...);
8      ...
9      // Contiguous write to storage
10     ADIO_WriteContig(...);
11     ...
12   }
13   ...
14 }
```

Listing 1: Pseudocode for MPI collective I/O write operations

```
1  ADIOI_R_Exchange_data(...)
2  {
3    MPI_Alltoall(...);
4
5    for (i=0; i < nprocs; i++) {
6        MPI_Irecv(...)
7    }
8
9    for (i=0; i < nprocs; i++) {
10       MPI_Isend(...)
11   }
12
13   MPI_Waitall(...)
14   ADIOI_Fill_user_buffer(...)
15   MPI_Waitall(...)
16 }
```

Listing 2: Pseudocode for data shuffling in MPI collective I/O

B. Profiling MPI Collective I/O

Based on the above analysis on the implementation of MPI collective I/O, we add timers to measure the performance of data shuffling (T_s) and read/write (T_{IO}) operations in each iteration of MPI collective I/O.

During profiling, we use the same five nodes as Section III-C. Among the five nodes, four of them run IOR and one works as a storage node. For IOR, we use 16 processes (4 processes per node), and set "segment count", "block size", and "transfer size" as 2, 512MB, and 16MB respectively. Total workload size for the four nodes is 16 GB. We use one aggregator per node. Table II shows our profiling results.

TABLE II: Profiling results for MPI collective I/O with IOR

Item	HDD	SSD	NVM
I/O time (s)	5938.91	1002.93	986.15
Shuffle time (s)	466.21	499.30	494.61
Ratio (shuffle time to collective I/O time)	7.85%	49.93%	50.16%
Average IO time per iteration (ms)	170.38	28.77	28.29
Average shuffle time per iteration (ms)	13.77	14.32	14.19

TABLE III: Notation of our performance modeling for MPI collective I/O

$T_{collective}$	The collective IO time.
$T_{individual}$	The individual I/O time.
T_{comm}	Data shuffling time.
T_{IO}	IO operation time.
T_{other}	Other performance cost besides data shuffling.
msg_size_i	The size of data that are communicated between the slowest aggregator and each MPI process for data shuffling in an iteration i.
τ	The ratio of data participated in data shuffling to total data.
$iter$	The number of iterations within the iterative collective I/O.
T_w	Communication time independent of the message size.
T_s	Communication time in proportion to the message size
bdw_{seq}	Sequential end-to-end I/O bandwidth.
bdw_{ran}	Random end-to-end I/O bandwidth.

The table reveals that from HDD, SSD, to NVM, the ratio of shuffle time to total collective I/O time increases from 7.85% to 50.16%. The shuffle time accounts for a larger percentage of performance loss, when we use NVM. Note that the shuffle time remains stable across the cases of HDD, SSD, and NVM, even through the ratio is different in the three cases. Because we use the same MPI implementation and the same I/O workload for the three cases, the communication pattern should be identical for the three cases and the shuffle time should be stable across the three cases.

C. Performance Modeling for MPI Collective I/O

We model MPI collective I/O performance based on the above discussion. The notation for our models is summarized in Table III.

MPI collective I/O ($T_{collective}$) is generally modeled in Equation 1. The equation includes the data shuffling time (T_{comm}), I/O operation time (T_{IO}), and other performance cost (T_{other}) because of the implementation of MPI collective I/O. T_{comm} and T_{IO} depend on data size and data access patterns of MPI processes. We model them as follows.

$$T_{collective} = T_{comm} + T_{IO} + T_{other} \quad (1)$$

Data shuffling time (T_{comm}) is modeled in Equation 2. T_{comm} is for one MPI aggregator (the slowest aggregator). There might be multiple aggregators involved in the collective I/O, but their data shuffling times are overlapped. The data shuffling phase iteratively sends or receives data between the aggregator and other MPI processes.

In Equation 2, at a specific iteration i, msg_size_i of data is communicated between the aggregator and each MPI process for data shuffling. In total, $\sum_{i=1}^{iter} msg\_size_i$ of data, which is the total amount of data from one MPI process for doing I/O operation, is communicated. There might be multiple MPI processes concurrently communicating with the aggregator shown in Lines 6 and 10 of Listing 2, but their communication times are overlapped. Note that it is possible that only a part of the total data is really communicated, while the other part of the data already reside in some aggregator and do not need to be transfered between the aggregators. To capture the above

fact, we introduce a parameter, τ. So, $msg\_size_i \times \tau$ is the amount of data that is really involved in the data shuffling between an MPI process and the slowest aggregator. Note that τ is application-dependent and related with the application's inherent I/O access pattern.

Based on the above discussion, the communication time for an iteration i is modeled by $T_s + T_w \times msg\_size_i \times \tau$, in which T_s represents the communication time unrelated with the message size, such as communication initialization time, and T_w represents the communication time related with the message size (or more precisely speaking, in proportion to the message size).

$$T_{comm} = \sum_{i=1}^{iter}(T_s + T_w \times msg\_size_i \times \tau) \quad (2)$$

I/O operation time (T_{IO}) is modeled in Equation 3. The numerator of the equation is the data ready for I/O operation. bdw_{seq} in the denominator is the end-to-end bandwidth (between the end of a compute node and the end of a storage node), and bdw_{seq} is the bandwidth for doing sequential I/O, because after data shuffling, there is supposed to be sequential data accesses between the aggregator and storage node.

$$T_{IO} = \frac{\sum_{i=1}^{iter} msg\_size_i}{bdw_{seq}} \quad (3)$$

T_{other} in Equation 1 is the other performance cost besides data shuffling, including memory mapping, variable initialization, system logs, and data checking for data alignment.

MPI individual I/O. To make a comparison between MPI collective I/O and individual I/O, we also model the performance of individual I/O, shown in Equation 4. $T_{individual}$ is much simpler than the collective I/O, because it does not have data shuffling, and I/O operations (T_{IO}) from each MPI process happen independently. To calculate T_{IO}, we use the end-to-end bandwidth for random data access (bdw_{ran}), shown in Equation 5. This is based on an assumption that data accesses from MPI processes are random without coordination as the collective I/O, but whether this assumption is true depends on the data access pattern of the application.

$$T_{individual} = T_{IO} + T_{other} \quad (4)$$

$$T_{IO} = \frac{\sum_{i=1}^{iter} msg\_size_i}{bdw_{ran}} \quad (5)$$

Model usage. To use the model, we need to know a set of parameters, including application-independent ones and application-dependent ones. The application-independent parameters include T_s, T_w, bdw_{seq}, bdw_{ran}, and T_{other}, which are measured only once on any platform. The application-dependent parameters include msg_size, τ, and number of iterations $iter$.

T_s and T_w are measured by running an MPI-based micro-benchmark doing ping-pong communication between compute node and storage node with different message sizes. We measure the communication time for each message size and use a linear regression to get T_s and T_w. In our test environment, $T_s = 5.39e-3$ (s) and $T_w = 3.35e-2$ (s/MB).

The parameters, bdw_{seq} and bdw_{ran}, can be measured by using IOR. In particular, we deploy IOR on our test environment with four compute nodes and one storage node. Using IOR, we perform read or write I/O operations for 2GB data. We set "reorder tasks to random" to enable either random or sequential I/O accesses with 16 MPI processes (4 processes per node), and then calculate bdw_{seq} and bdw_{ran}. Table IV summarizes the results in our test platform. One interesting observation is that between SSD and NVM, there is no big difference in terms of bdw_{seq} and bdw_{ran}, shown in the table. This is because of the fact that SSD and NVM have a larger device bandwidth than HDD, such that the end-to-end bandwidth is limited by networking.

TABLE IV: bdw_{seq} and bdw_{ran} in our test platform.

	HDD	**SSD**	**NVM**
bdw_{seq} (MB/s)	58.11	110.98	112.31
bdw_{ran} (MB/s)	26.72	101.86	110.51

T_{other} is assumed to be constant in our model, and can be measured through IOR as well. In particular, we deploy the same tests as the ones for measuring bdw_{seq} and bdw_{ran}, and measure $T_{individual}$, $T_{collective}$, I/O operation time and shuffling time. Then, we calculate T_{other} based on Equations 1 and 4 for collective I/O and individual I/O, respectively. In our tests, we find that T_{other} is much smaller than I/O operation time and data shuffling time. Hence we set T_{other} as zero during model validation (Section IV-D).

The total data size (see the discussion on $\sum_{i=1}^{iter} msg\_size_i$ in the MPI collective I/O modeling) can be obtained by examining the application, particularly MPI I/O calls (e.g., MPI_File_write_all() and MPI_File_read_all()). In each iteration, msg_size is constant in our model, which is equal to the collective buffer size (16MB in our tests) in ROMIO. The number of iterations ($iter$) is equal to the total data size divided by the constant collective buffer size.

The parameter τ depends on the application I/O access pattern and MPI implementation. It is challenging to predict or choose a universally appropriate value for all possible cases. Also, it is challenging to ask the user to quantify their workload characteristics and choose τ. For simplicity, assuming that each MPI process needs to do the same size of IO, and during the two phases (data shuffling and I/O phases) of collective IO, the data sent from all non-aggregator MPI processes are evenly handled by the aggregators, then we roughly estimate τ as the number of non-aggregator MPI processes divided by total number of MPI processes.

For example, suppose that we have 8 processes in total, 2 of them are I/O aggregators, and each process needs to write 1MB data (i.e., 8MB for 8 processes). Then during the two phases of collective IO, the 6 non-aggregators need to send totally 6MB data to the 2 aggregators. Based on the definition of τ, τ = 6MB/8MB = 75%. Based on our estimation of τ, τ = (#non-aggregators / total number of MPI processes) = $6/8 = 75\%$. Our estimation of τ has a great match to the real value of τ.

TABLE V: Comparison of estimated and measured I/O times with 4 compute nodes (4 processes per node). The percentage numbers in brackets are prediction errors.

Device	HDD	SSD	NVM
Collective I/O estimated time (s)	411.78(6.7%)	286.21(3.2%)	284.46(14.4%)
Collective I/O Measured time (s)	385.86	277.46	242.54
Individual I/O estimated time (s)	613.17(3.4%)	160.84(9.8%)	145.88(3.2%)
Individual I/O Measured time (s)	593.04	146.50	146.35

TABLE VI: Comparison of estimated and measured I/O times with 2 compute nodes (8 processes per node). The percentage numbers in brackets are prediction errors.

Device	HDD	SSD	NVM
Collective I/O estimated time (s)	350.90(1.04%)	216.58(0.53%)	214.83(0.85%)
Collective I/O Measured time (s)	354.59	217.74	213.01
Individual I/O estimated time (s)	613.17(5.66%)	160.84(9.86%)	145.88(0.46%)
Individual I/O Measured time (s)	580.32	146.40	146.55

Note that when estimating τ, we assume that the data sent from all non-aggregator MPI processes are evenly handled by the aggregators. In practice, the data can be unevenly handled by the aggregators. It is even possible that some aggregator does not need to do any data shuffling. However, our model is for the slowest aggregator that has the longest data shuffling time and dominates data shuffling time of all aggregators. τ for the slowest aggregator can be estimated well by our method in most cases.

Discussion. Our model has two limitations. First, we do not distinguish intra- and inter-node communication in Equation 2 when modeling data shuffling time. In particular, we measure T_s and T_w based on inter-node communication and use them in Equation 2, no matter whether data shuffling happens within a node or between nodes. Second, we assume that the data shuffling times of all aggregators can be greatly overlapped. However, depending on data access patterns of MPI processes, different aggregators working with different MPI processes can have different, non-overlapped data shuffling time.

To fix the above model limitation, we must have good knowledge on the execution environment, such that we know how MPI processes are mapped to nodes in order to determine intra- and inter-node communication; we must also have deep knowledge on data access patterns of each MPI process. However, having the above knowledge greatly limits the model usability and generality, while providing limited help for modeling accuracy. Hence, we do not assume such knowledge is available in our model. Our results show that the current model works reasonably well.

D. Model Validation

We verify our model accuracy with IOR. We test two cases, one with 4 compute nodes (4 processes per node) and the other with 2 compute nodes (8 processes per node). Both cases have one storage node. For IOR, "segment count", "block size", and the collective buffer size are set as 2, 64MB, and 16MB respectively. We use one aggregator per node in validation tests.

Tables V and VI show the validation results. In general, our model achieves high accuracy in 12 validation tests (average error 4.93% and at most 14.4%). More importantly, our model correctly captures performance trend across the three devices in different cases.

E. Model Implication

Our model enables us to explore the tradeoff between data shuffling cost and collective I/O benefit in a variety of environments with different storage devices. Hence it can be used to enable adaptive performance optimization and improve I/O performance for the future HPC using NVM-based storage.

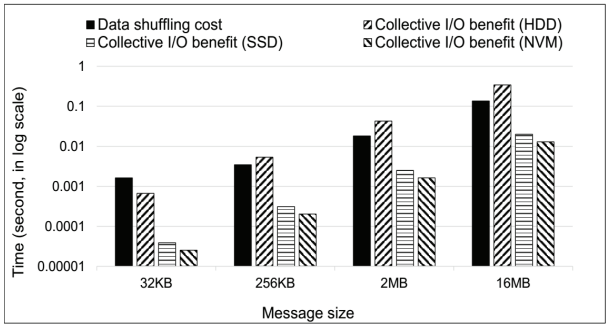

Fig. 11: Explore the performance tradeoff between data shuffling cost and collective I/O benefit.

As a case study, we use our model to study the tradeoff between data shuffling cost (Equation 2) and collective I/O benefit. The collective I/O benefit is quantified by $(T_{individual} - T_{collective})$. We focus on one iteration (i.e., $iter = 1$) and change the message size. We use bandwidth and communication parameters (i.e., T_w and T_s) measured in our platform for our study. Figure 11 shows the result, assuming that there are 4 compute nodes, 1 storage node, and 4 MPI processes per node.

The figure reveals that both data shuffling cost and collective I/O benefit increase as the message size increases, but at different rates. For HDD, although the data shuffling cost is larger than the benefit when the message size is small (32KB), the data shuffling cost is smaller than the benefit when the message size is large (2MB and 16MB). However, for SSD and NVM, the data shuffling cost is always larger than the benefit, which explains why collective I/O performs consistently worse than individual I/O in Tables V.

V. Related Work

Non-volatile memory. Prior NVM studies can be roughly classified into several categories. Some earlier studies focus on the architecture-level design issues of NVM [16], [17], [18], [19], such as wear-leveling, read-write disparity issues, etc. Most of these studies consider NVM as a displacement of DRAM at the architecture level. Another alternative is to consider NVM as a storage device, such as Onyx [20], Moneta [21], and PMBD [14]. The recently announced Intel Optane product [4] also falls into this category. Researchers have also studied on the system and application level support for NVM. Some prior studies have explored file systems for NVM. For example, BPFS [22] uses shadow paging

techniques for fast and reliable updates to critical file system metadata structures. SCMFS [23] adopts a scheme similar to page table in memory management for file management in NVM. PMFS [24] allows to use memory mapping (mmap) for directly accessing NVM space and avoids redundant data copies. In order to take advantage of byte-addressability and persistency of NVM, a large body of research on NVM is on developing new programming models for NVM. For example, Mnemosyne [6] gives a simple programming interface for NVM, such as declaring non-volatile data objects. CDDS [7] attempts to provide consistent and durable data structures. NV-Heaps [8] gives a simple model with support of transactional semantics. SoftPM [25] offers a memory abstraction similar to `malloc` for allocating objects in NVM. In this study we treat NVM as a storage device and deploy conventional file systems atop for HPC applications. Our observations have confirmed that the high-speed NVM could significantly improve HPC application performance, however, the end-to-end effect is workload dependent and related to a variety of factors in the I/O stack.

MPI I/O. ROMIO [26] is a widely used implementation of MPI-IO, which is included in the MPICH library. ROMIO uses two-phase I/O strategy [27] for collective I/O. Some prior work explores the determination of optimal number of aggregators for MPI I/O [28]. Some prior work takes into account the network topology for deciding aggregators and introduces an optimized buffering system to reduce the aggregation cost [27]. Another study performs collective I/O while retaining access patterns of MPI processes before collective I/O to enable better cache management [29]. Our work is different from the prior studies by considering the impact of NVM on MPI I/O.

VI. Conclusions

We study the impact of upcoming NVM on HPC I/O. Given distinct performance characteristics of NVM, the existing I/O stack must be re-examined to optimize performance. Through our comprehensive performance study and modeling, we reveal the diminishing benefits of page cache, ignorable overhead of MPI individual I/O, and inappropriate performance optimization of MPI collective I/O. Our work lays some foundation for the deployment of NVM in the future HPC.

Acknowledgement. This work is partially supported by U.S. National Science Foundation (CNS-1617967, CCF-1553645, CCF-1453705, CCF-1629291) and Louisiana Board of Regents under Grant LEQSF(2014-17)-RD-A-01.

References

[1] B. C. Lee, P. Zhou, J. Yang, Y. Zhang, B. Zhao, E. Ipek, O. Mutlu, and D. Burger, "Phase-Change Technology and the Future of Main Memory," *IEEE Micro*, vol. 30, no. 1, pp. 143–143, 2010.

[2] B. Dieny, R. S. G. Prenat, and U. Ebels, "Spin-dependent Phenomena and Their Implementation in Spintronic Devices," in *International Symposium on VLSI Technology, Systems and Applications*, 2008.

[3] K. Suzuki and S. Swanson, "The Non-Volatile Memory Technology Database (NVMDB)," Department of Computer Science & Engineering, University of California, San Diego, Tech. Rep. CS2015-1011, 2015, http://nvmdb.ucsd.edu.

[4] Intel, https://www.intel.com/OptaneMemory.

[5] Https://github.com/linux-pmbd/pmbd.

[6] H. Volos, A. J. Tack, and M. M. Swift, "Mnemosyne: Light Weight Persistent Memory," in *Architectural Support for Programming Languages and Operating Systems*, 2011.

[7] S. Venkataraman, N. Tolia, P. Ranganathan, and R. H. Campbell, "Consistent and Durable Data Structures for Non-volatile Byte-Addressable Memory," in *USENIX Conference on File and Storage Technologies (FAST 2011)*, 2011.

[8] J. Coburn, A. M. Caufield, A. Akel, L. M. Grupp, R. K. Gupta, R. Jhala, and S. Swanson, "NV-Heaps: Making Persistent Objects Fast and Safe with Next-Generation, Non-Volatile Memory," in *Architectural Support for Programming Languages and Operating Systems*, 2011.

[9] National Energy Research Scientific Computing Center, http://www.nersc.gov/about/groups/advanced-technologies-group/benchmark-software/benchmark-applications/the-nersc-madbench-benchmark/.

[10] ——, http://www.nersc.gov/users/computational-systems/cori/nersc-8-procurement/trinity-nersc-8-rfp/nersc-8-trinity-benchmarks/ior/.

[11] CORAL Benchmark Codes, https://asc.llnl.gov/CORAL-benchmarks/.

[12] Avery Ching, http://users.eecs.northwestern.edu/.

[13] "mpiBLAST: Open-Source Parallel BLAST," http://www.mpiblast.org/.

[14] F. Chen, M. P. Mesnier, and S. Hahn, "A Protected Block Device for Persistent Memory," in *International Conference on Massive Storage Systems and Technology*.

[15] R. Thakur, W. Gropp, and E. Lusk, "Data sieving and collective i/o in romio," in *The Seventh Symposium on the Frontiers of Massively Parallel Computation*, 1999.

[16] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger, "Architecting Phase Change Memory as a Scalable DRAM Alternative," in *Proceedings of the 36th International Symposium on Computer Architecture (ISCA 2009)*, 2009.

[17] M. K. Qureshi, J. Karidis, M. Franceschini, V. Srinivasan, L. Lastras, and B. Abali, "Enhancing Lifetime and Security of PCM-based Main Memory with Start-gap Wear Leveling," in *International Symposium on Microarchitecture*, Dec 2009.

[18] M. K. Qureshi, V. Srinivasan, and J. A. Rivers, "Scalable High Performance Main Memory System using Phase-Change Memory Technology," in *Proceedings of the 36th International Symposium on Computer Architecture (ISCA 2009)*, June 2009.

[19] P. Zhou, B. Zhao, J. Yang, and Y. Zhang, "A Durable and Energy Efficient Main Memory Using Phase Change Memory Technology," in *Proceedings of the 36th International Symposium on Computer Architecture (ISCA 2009)*, June 2009.

[20] A. Akel, A. M. Caulfield, T. I. Mollov, R. K. Gupta, and S. Swanson, "Onyx: A Prototype Phase Change Memory Storage Array," in *Proceedings of the 3rd USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 2011)*, Portland, OR, June 14 2011.

[21] A. M. Caulfield, A. De, J. Coburn, T. I. Mollov, R. K. Gupta, and S. Swanson, "Moneta: A High-Performance Storage Array Architecture for Next-generation, Non-volatile Memories," in *Proceedings of the 43rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2010)*, Atlanta, Georgia, Dec 4-8 2010.

[22] J. Condit, E. B. Nightingale, C. Frost, E. Ipek, D. Burger, B. C. Lee, and D. Coetzee, "Better I/O Through Byte-Addressable, Persistent Memory," in *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP 09)*, Big Sky, MT, October 2009.

[23] X. Wu and A. L. N. Reddy, "SCMFS: A File System for Storage Class Memory," in *Proceedings of Supercomputing*, 2011.

[24] S. R. Dulloor, S. Kumar, A. Keshavamurthy, P. Lantz, D. Reddy, and R. S. J. Jackson.

[25] J. Guerra, L. Mármol, D. Campello, C. Crespo, R. Rangaswami, and J. Wei, "Software Persistent Memory," in *Proceedings of the 2012 USENIX Annual Technical Conference*, Boston, MA, June 13-15 2012.

[26] R. Thakur, W. Gropp, and E. Lusk, "A Case for Using MPI's Derived Datatypes to Improve I/O Performance," in *ACM/IEEE Conference on Supercomputing*, 1998.

[27] F. Tessier, P. Malakar, V. Vishwanath, E. Jeannot, and F. Isaila, "Topology-Aware Data Aggregation for Intensive I/O on Large-Scale Supercomputers," *The Workshop on Optimization of Communication in HPC*, pp. 73–81, 2016.

[28] M. Chaarawi and E. Gabriel, "Automatically Selecting the Number of Aggregators for Collective I/O Operations," in *International Conference on Cluster Computing (Cluster)*, 2011.

[29] J. Liu, Y. Chen, and S. Byna, "Collective Computing for Scientific Big Data Analysis," in *International Conference on Parallel Processing Workshops (ICPPW)*, 2015.