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

by

Wei Liu

A thesis submitted in partial satisfaction of the

requirements for the degree of

Master Science

in

IGP, Electrical Engineering and Computer Science Emphasis, M.S.

in the

Graduate Division

of the

University of California, Merced

Committee in charge:

Professor Dong Li, Chair
Professor Mukesh Singhal
Professor Sungjin Im

Spring 2017

The thesis of Wei Liu, titled Performance Evaluation and Modeling of HPC I/O on Non-Volatile Memory, is approved:

Chair _____ Date _____
Dong Li

_____ Date _____
Mukesh Singhal

_____ Date _____
Sungjin Im

University of California, Merced

Contents

List of Figures

List of Tables

Acknowledgments

Thanks for Dr. Dong Li's help throughout my master student life. Thanks for my lab mates for being as a family. Thanks my parents for your love, I love you too.

Abstract

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

by

Wei Liu

Master Science in IGP, Electrical Engineering and Computer Science Emphasis, M.S.

University of California, Merced

Professor Dong Li, Chair

Modern HPC applications pose high demands on I/O performance and storage capability. The emerging non-volatile memory (NVM) techniques, such as Phase Change Memory and STT-RAM, offer low-latency, high bandwidth, and persistence for HPC applications. However, the existing I/O stack, including OS, high level library, I/O middleware, and applications, are designed and optimized based on an assumption of disk-based storage. To effectively use NVM, we must re-examine the existing I/O sub-system 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 of 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 caches; (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 caches, ignorable 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 model the performance of MPI collective I/O and study the complex interaction between data shuffling, storage performance, and I/O access patterns. Extensive experiments have been conducted to verify our analysis.

Keywords: NVM, page cache, MPI I/O, Collective I/O

Chapter 1

Introduction

Modern 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 to study Q continuum works on 2PM per simulation. Both of these simulations require transformation of the data representation, which pose high demands on I/O performance and storage capability.

The emerging non-volatile memory (NVM [18]) techniques, such as Phase Change Memory (PCM) [16] and STT-RAM [11], offer low-latency access, high bandwidth, and persistence. Their performance is much better than the traditional hard drive, and close to or even match that of DRAM. The non-volatility and high performance of NVM blurs the line between storage and main memory, hinting at opportunities to overhaul classical IO system and memory hierarchies. Table 1.1 summarizes the characteristics of different NVM

technologies and compares them to traditional DRAM and storage technologies.

Table 1.1: Memory Technology Summary[24]

	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. 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 (e.g., hard drive) 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 such a block-based NVM model is likely to become the mainstream industry (e.g., the recently announced Intel Optane[14]) and be adopted in the near future soon . We pose the following questions to gain insights into the application performance with NVM.

- What is the impact of NVM on the performance of the traditional page caches? Is it still reasonable to use page caches for NVM-based storage?

- Comparing MPI individual I/O and POSIX I/O based on NVM, what are their performance in the HPC domain? With a high-speed NVM device, would such an additional layer bring too much overhead?

- MPI I/O introduces collective I/O based techniques to optimize application performance, based on the assumption of bad 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 PMBD[6]. 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 the 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 the page cache, NVM does not exhibit significant performance advantages over SSD and hard drive. For example, when 2GB HACC benchmark working on 11GB page cache, performance difference has only 4% difference among three kinds devices.

- MPI individual I/O and POSIX I/O have ignorable 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 I/O cases. This indicates that given the current highly optimized implementation of MPI individual I/O, the performance overhead of MPI individual I/O is still not 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 good performance 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 38.39% 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 trade off 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.

sectionOur Contribution Several contributions are made in this paper: (1). We investigate the application level performance impact of page cache by controlling page cache size.

Page cache plays an important role in HPC. With page cache, highly used data could be buffered so less cache miss penalty could be achieved. Bigger page cache will lead to more data be buffered and better performance could be reach. As for high speed NVM storage device, we use experiments to find out page cache still play an important role in speed up performance. (2). MPIIO is a highly used parallel computing architecture standard. This parallel computing standard is aim to improve IO speed by synchronously doing multiple IO threads. We use several comparison experiments to check whether this standard still take a great infect in high speed NVM storages. (3). Collective IO is another technology used in HPC to speed up parallel IO performance. Several profiling works are done in this article to check whether collective IO is useful in high speed performance NVM. Furthermore, we made a model to check if this speed up is suitable for any circumstance and in which environment collective IO will take its effect.

The rest of the paper proceeds as follows. Chapter II covers the background, including NVM emulator, applications, and preliminary MPI I/O information. In Chapter III, we present performance characteristics of applications under various test environment. In Chapter IV, we introduce our performance model for the MPI collective I/O. In Chapter V, we discuss related work and conclude in Section VII.

Chapter 2

Background

2.1 NVM Usage Model

Non-volatile memory (NVM), represented by Phase Change Memory (PCM) and Spin-transfer-torque RAM (STT-RAM), is a pivotal technology, providing a variety of attractive technical features, such as low power consumption, high endurance, and byte addressability, DRAM-like access speed, disk-like persistence, etc. Drawing a blurry line between traditional volatile memory and persistent storage, NVM has at least two basic usage models as follows.

(1) **Memory-based Model**. NVM is treated as the regular, byte-addressable, DRAM-based 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. For example, applications can declare certain in-memory object non-volatile. 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 [29], CDDCS [28], 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 new memory 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, and 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 the 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 employing this block device model [14]. In

this work, We assume a storage-based model in this work, which is the most promising NVM solution in the near future.

2.2 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 patterns (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 considered as the worst situation for disk drives, because this situation creates a disk head's "seek storm" and lose performance. 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 scatting and gathering 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 all MPI processes. Figure 2.1 depicts the MPI collective I/O scheme. 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.

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

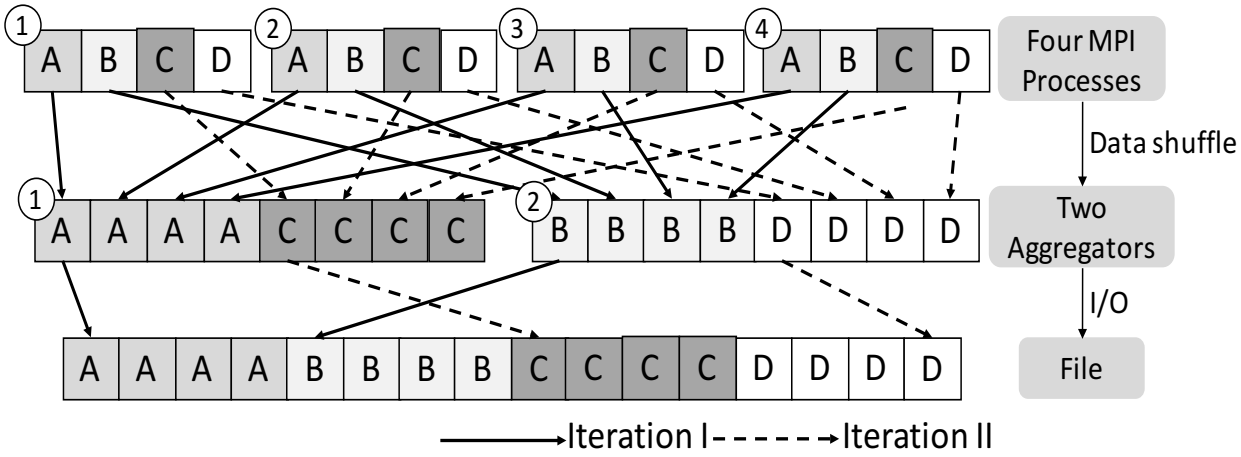

Figure 2.1: MPI collective I/O scheme. The numbers in circles are MPI process IDs. There are two aggregators (MPI processes 1 and 2) in this example.

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 the performance benefits of optimized I/O operations from MPI collective I/O (see Figure 2.1). However, with high-speed storage device, such as NVM and SSD, which are relatively insensitive to I/O patterns (e.g., random accesses) and carry much higher I/O performance, MPI collective I/O may not remain advantageous, especially considering the involved data shuffling overhead. 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. In this paper, we will particularly study this issue.

2.3 Benchmarks

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

MADBench2.

This benchmark is a "stripped-down" version of MADCAP (a Microwave Anisotropy Dataset Computational Analysis Package) [20]. 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. It involves some parameters setting the workload pattern, including "NO_PIX" to specify the number of pixels, "NO_BIN" to specify number of multiple bins, "NO_GANG" to specify the number of independent work gangs to divide the processors into, and "BLOCKSIZE" to specify the size per block used in ScaLAPACK operations.

IOR

IOR is a benchmark widely used to study parallel I/O performance at both the POSIX and MPI-IO level [21]. 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 configuration parameters are related to our work, including "segment count", "block size", and "transfer size", shown in Figure 2.2. Given some data for doing collective

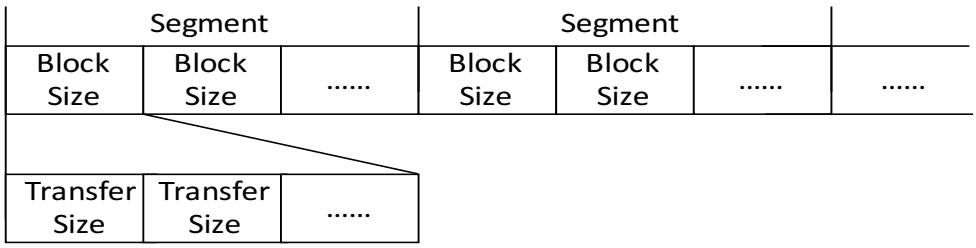

Figure 2.2: The data transf from IOR benchmark

I/O, the data 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 receive at most "transfer" size of data from another MPI process. IOR also has a parameter "reorder tasks to random", which enable random I/O accesses. We use this option for IOR throughput the paper.

HACC-IO

This benchmark is the I/O kernel of HACC (an HPC application based on N-body simulation) [10]. 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 number of data to write is the "numparticles" multiplied by the number of MPI processes.

S3aSim.

This benchmark is an MPI-IO based sequence similarity search algorithm framework [2]. S3aSim emulates IO access patterns in mpiBLAST [19], which is "streaming-like", read-only data accesses. S3aSim has five working phases, and we focus on the I/O phase of this benchmark.

2.4 PMBD Emulator

As NVM devices are not available yet, we use Persistent Memory Block Driver (PMBD) [7], which is an 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, PMBD emulator unit has no difference from other physical block devices, while it provides configurable features of NVM devices, such as emulating various bandwidths, latencies, protections, etc.

Because we assume NVM will be exposed to OS as a contiguous ranged physical device.

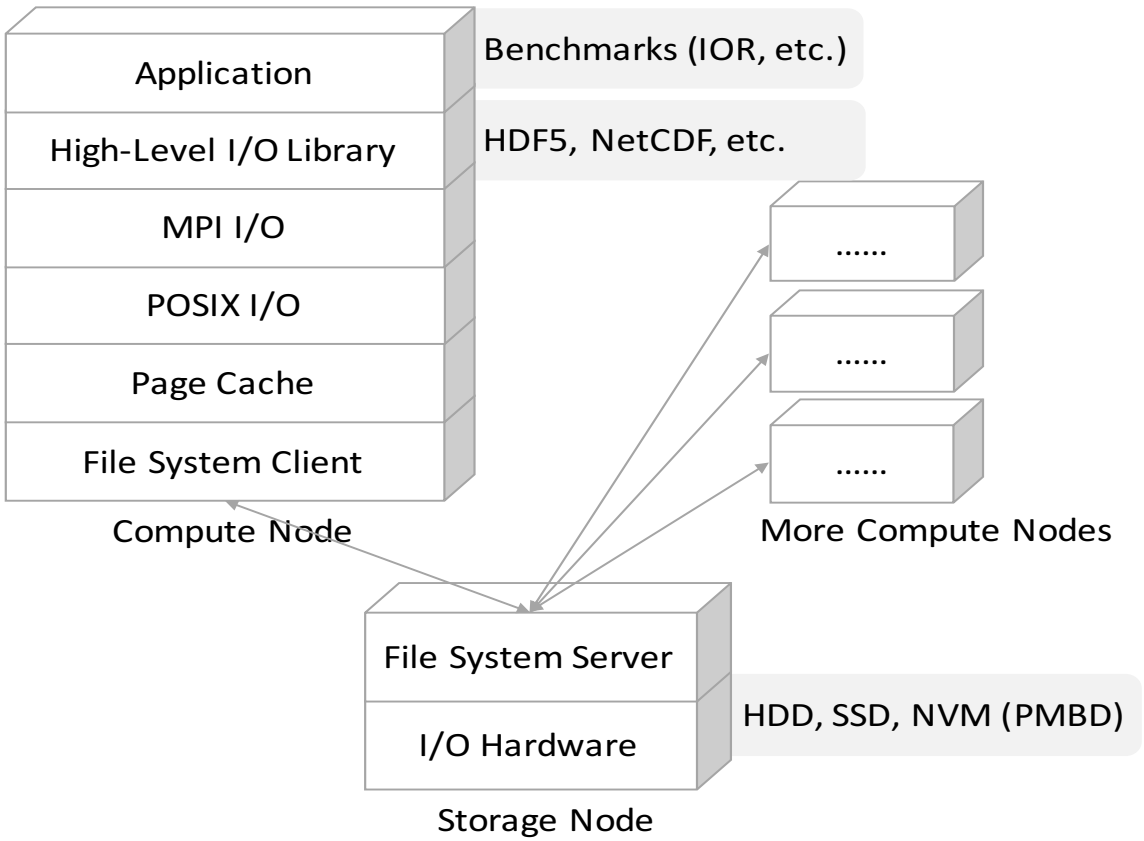

Figure 2.3: System Hierarchy

The mechanism of PMBD driver is mapping NVM physical pages into the kernel virtual address space to make it accessible. For each I/O request, driver will translate read/write request into responding load/store instruction to the physical address mapped.

2.5 I/O Hierarchy

The I/O stack in a typical HPC system has multiple layers, shown in Figure 2.3. 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 memory. Due to its "filtering" effect, the page cache can have a strong impact on IO 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 experiments, we have tested on both Network File System (NFS) and local file systems. MPI I/O built on top of POSIX I/O enables coordinated and remote I/O accesses for MPI processes.

Chapter 3

Performance Study

We present our performance analysis results in this section. We deploy our tests in a local cluster 3.1. 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.

NVM device is emulated by PMBD simulator. Besides PMBD, we used another two kinds storage device, HDD and SSD, as comparison. So that we have three different steps of I/O speed. In this article, we attached a separated HDD and SSD device into our main storage node and mounted as an extra block space. PMBD were also mounted as an spare

Table 3.1: Hardware configurations

Hardware	**LBNL Edison**	**Local clusters**
Nodes	4	4
DRAM	DDR3 64GB	DDR4 32GB
CPU	Intel(R) Ivy bridge	Intel(R) Xeon(R) CPU E5-2630 v3 @2.40GHz
Cache: L1-L2-L3	32K-256K-30M	32K-256K-20M
Privilege access	NO	YES

space into main node. In this way, we could run our experiments in these devices without any other system infects.

3.1 Impacts 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 improve performance. We measure performance of the three I/O devices with different page cache configurations and study the impact of page cache on application performance.

We use three benchmarks in our tests, HACC-IO, MADBench2, and S3aSim. Those 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 3.1, 3.2, and 3.3 show the result for HACC-IO, MADBench2, and S3aSim respectively.

HACC-IO in Figure 3.1 simulates 13,107,200 particles in total (i.e., numparticles=13,107,200).

It computes and then generates about 2GB data, 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 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 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 and SSD is reduced by 92.7% and 84.8% respectively, while the performance loss with PMBD is only 11.5%. This example well illustrates that with high-speed NVM, the effect of page cache is weakened.

MADBench2 in Figure 3.2 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 for the phase W, which includes both read and write operations. MADBench2 tells us a story slightly different from HACC-IO. Because MADBench2 has a larger working set size, MADBench2 on all of the three block devices has performance loss. The page cache is unable to effectively cache all data, including those for HACC-IO and system. However, NVM performs the best in cases, because of its higher I/O bandwidth.

S3aSim in Figure 3.3 uses a working set size of 2GB (with 100 total query number, max size of each query as 5000, max count of each query as 10000). Comparing the performance of MADBench2 and S3aSim, we find that they have the same performance trend: the NVM

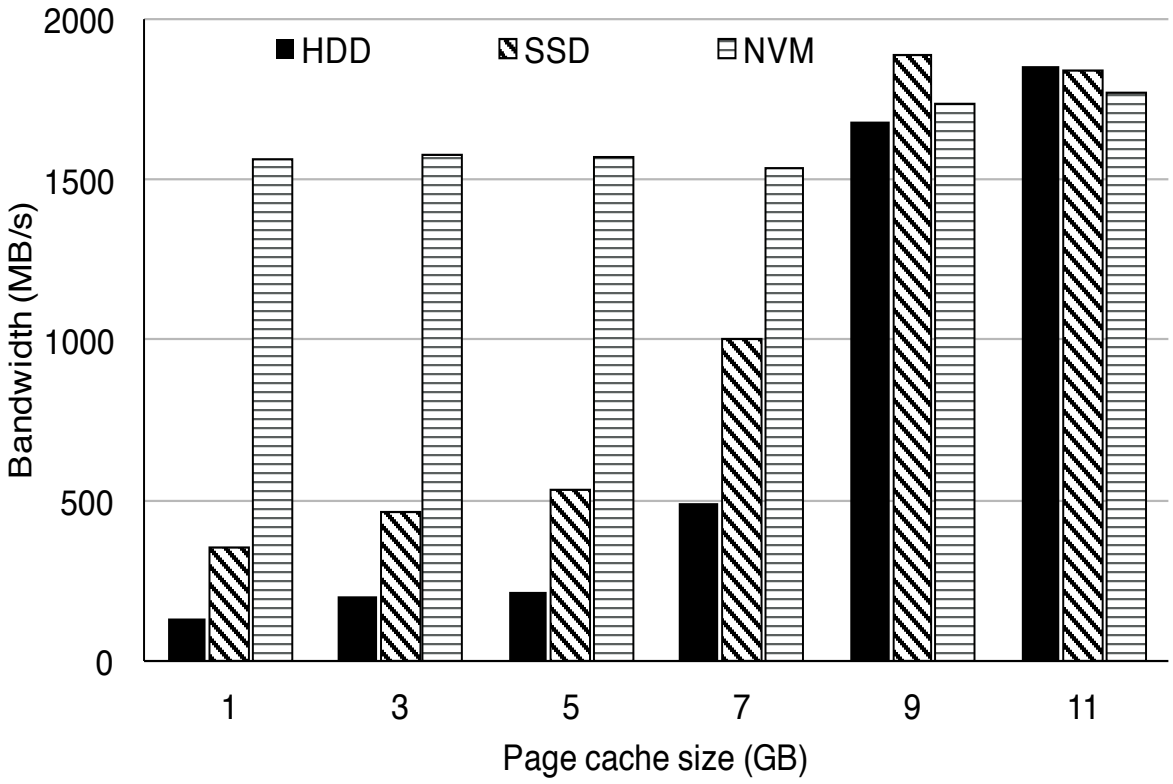

Figure 3.1: The performance study for the impacts of page cache on HACC-IO.

has the best performance in all cases, but their performance is different when the page cache size is small (1GB). For MADBench, NVM has significant performance reduction, 40.16% from 3GB to 1GB of the page cache size; for S3aSim, this performance degradation is only 5.91%. We attribute such performance difference in the performance loss to the 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 MADBench, the page cache takes effect, although the caching effects of page cache becomes smaller, when the page cache size is small (1GB).

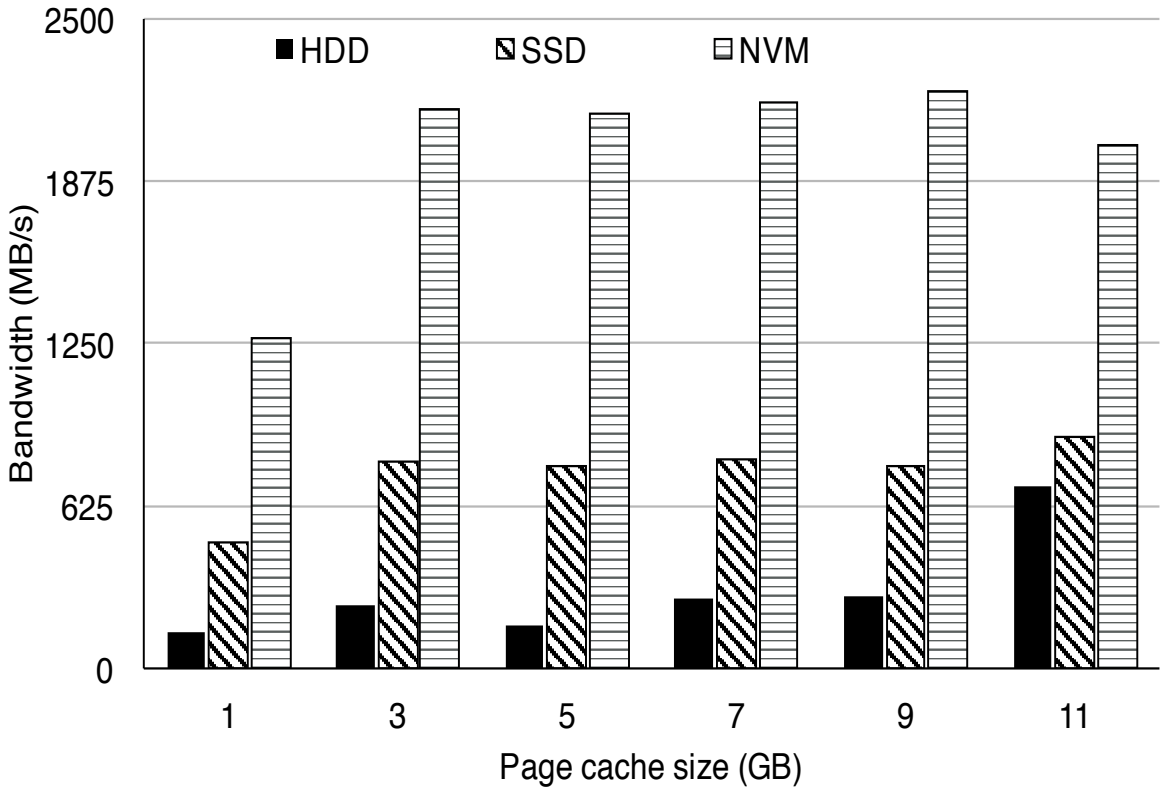

Figure 3.2: The performance study for the impacts of page cache on MADBench2.

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

Our study has 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 the page cache for certain workloads on NVM-based block device, which will save the limited page cache space for other system data, which

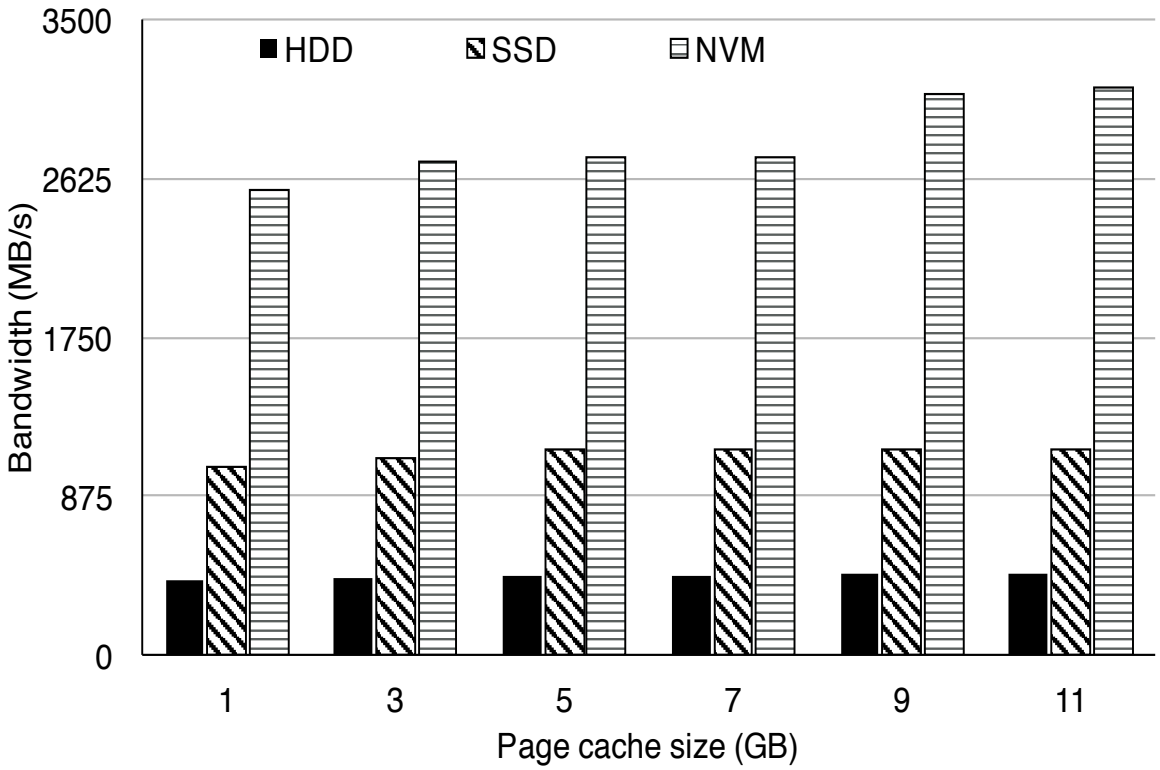

Figure 3.3: The performance study for the impacts of page cache on S3aSim.

in turn improves the performance of the whole system.

3.2 POSIX I/O and MPI Individual I/O

MPI I/O is built on top of POSIX I/O, shown in Figure 2.3, and 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 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 inevitably introduces 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 3.4 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 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 process in total), and the fifth node works as a storage node where the other four nodes remotely perform I/O operations. Hence, different from the tests on a single node, such deployment has the involvement of communication between the four nodes

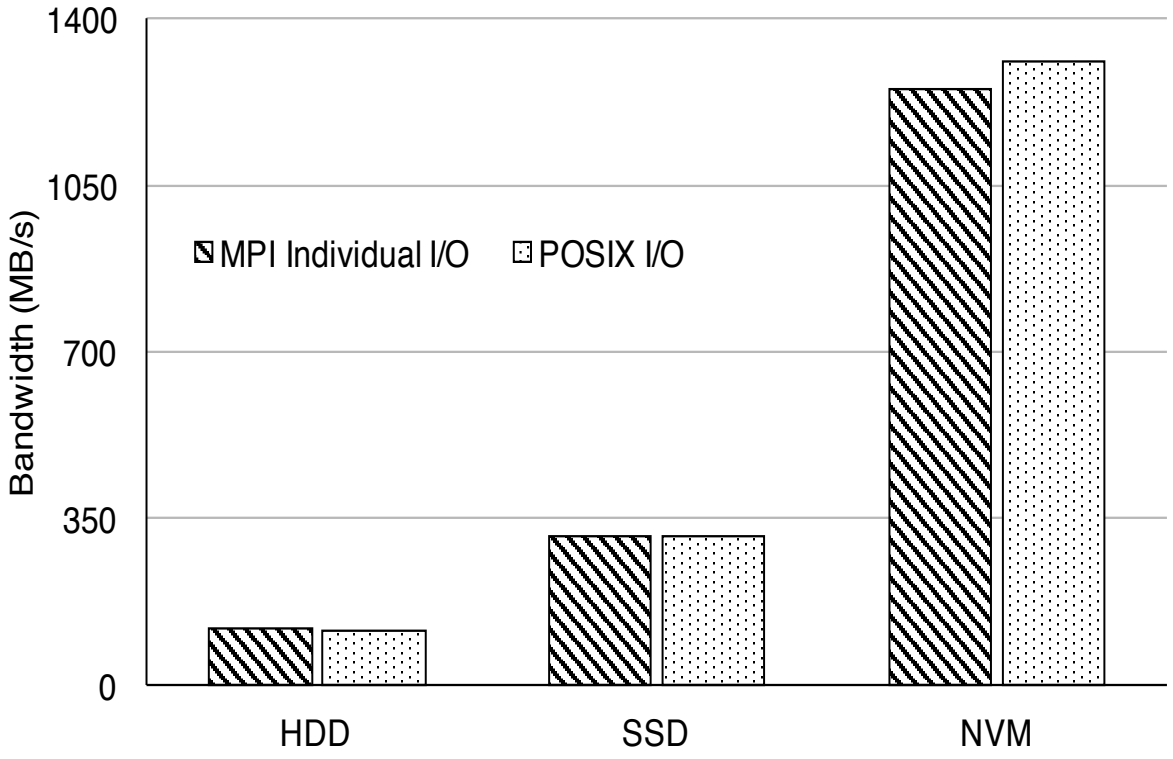

Figure 3.4: Comparing the performance of MPI individual I/O and POSIX I/O performance on a single node with IOR.

and the storage node. With such deployment, POSIX I/Os are performed with NFS in our test environment. Figure 3.5 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 one of major performance bottlenecks, 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

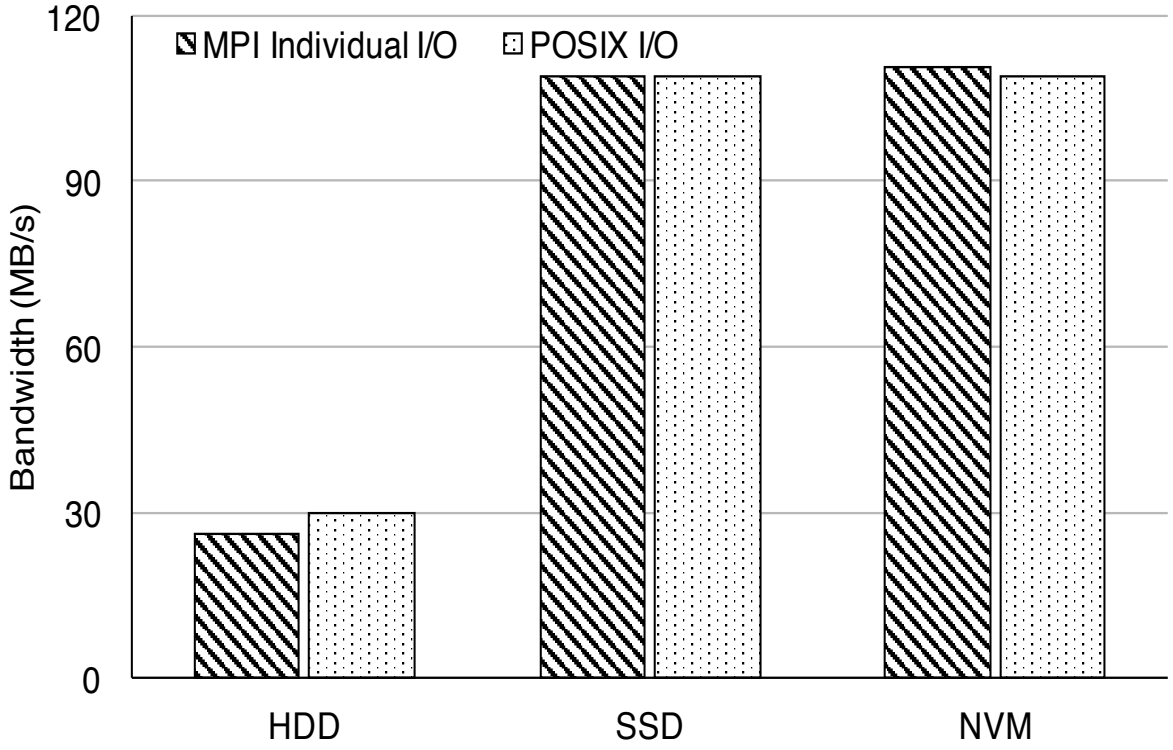

Figure 3.5: Comparing the performance of MPI individual I/O and POSIX I/O performance on multiple nodes with IOR.

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. In a multi-node environment, MPI individual I/O has ignorable performance overhead, even if we use NVM. This seem to indicate that the current implementation of MPI individual I/O is good for the future HPC equipped with the emerging NVM.

3.3 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 written 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 2.2. 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 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 use the same IOR configuration (including workload size, block size, and data transfer size) as that for MPI individual I/O and POSIX I/O (Section 3.2). We use five nodes for the tests, four of which runs the IOR benchmark. 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 3.6 and 3.7 show results for the case of 4 processes per node (16 processes in total) and 16 processes per node (64 process in total).

The figures reveal that SSD and NVM achieve better performance with MPI individual I/O than with MPI individual I/O. On the contrary, HDD benefits from the optimization with MPI collective I/O. We observed similar effects with 64 processes on 4 nodes, which

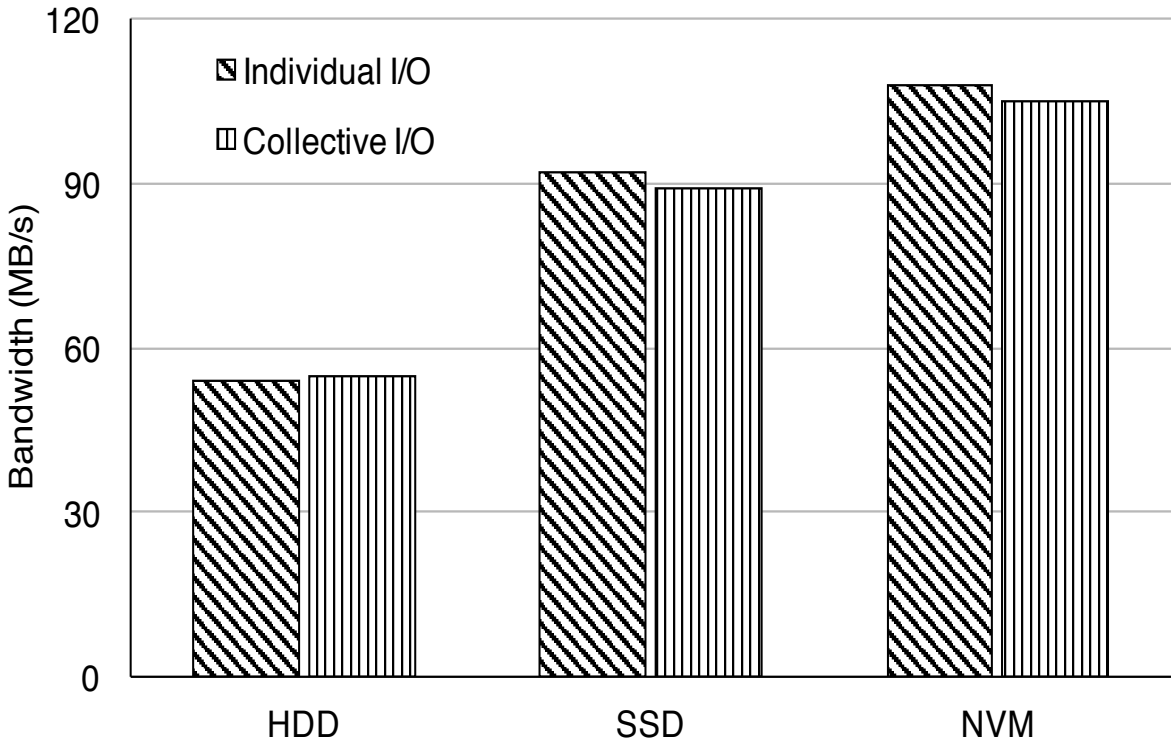

Figure 3.6: Comparing the performance of MPI collective I/O and MPI individual I/O (4 processes per node) with IOR.

introduces more intensive I/O operations.

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 also notice that the performance difference between MPI collective I/O and individual I/O becomes bigger in the case of 16 processes per node than in the case of

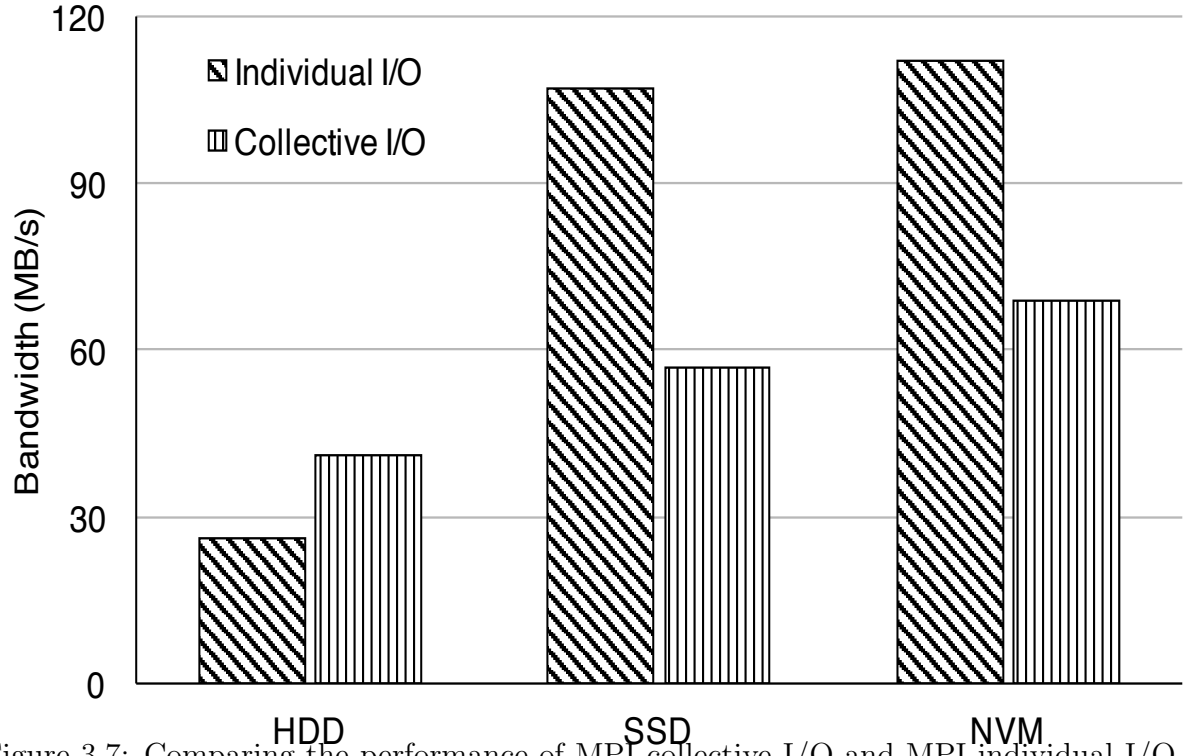

Figure 3.7: Comparing the performance of MPI collective I/O and MPI individual I/O (16 processes per node) with IOR.

4 processes per node. Such larger performance difference is due to the higher data shuffling cost when dealing with a large number of concurrently running processes.

3.4 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

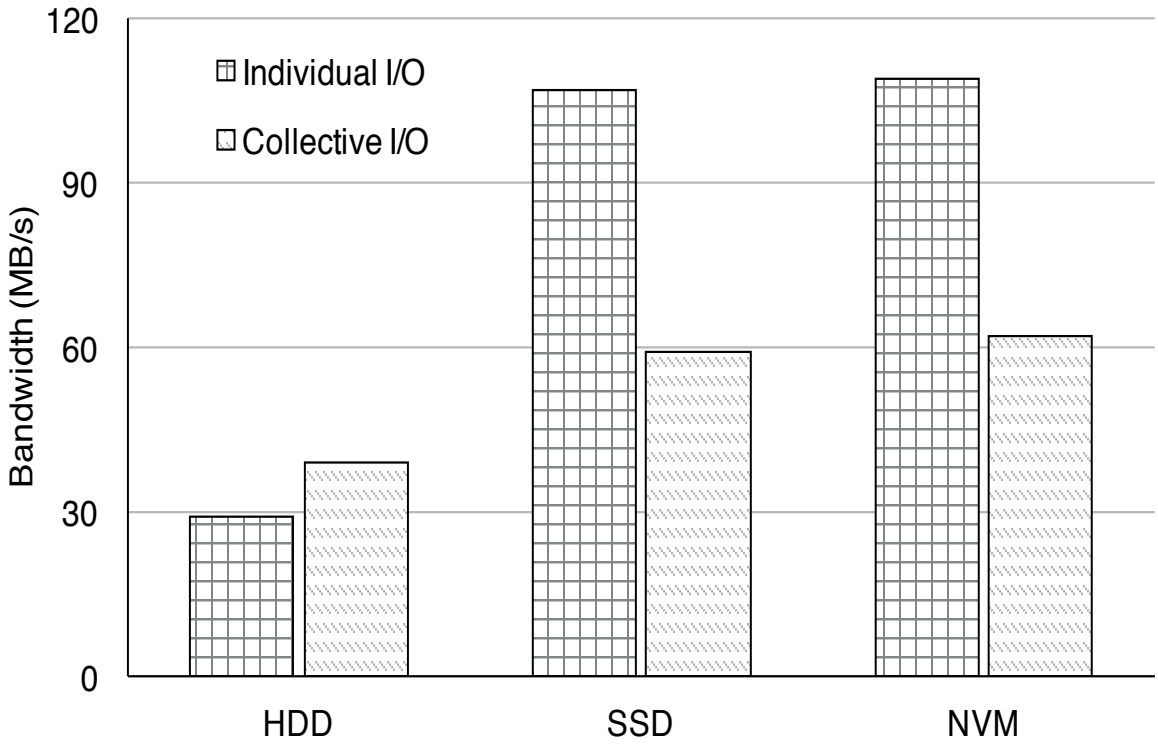

Figure 3.8: IOR Individual I/O and Collective I/O Performance Difference (64 Processors)

for NVM. This urges us to also revisit the existing mechanisms, besides MPI collective I/Os, that aim to optimize performance based on the 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 the next section, we further study the performance of MPI collective I/O and investigate why MPI collective I/O has worse performance. We also introduce a performance model that facilitates to make a decision on when to use MPI collective I/O.

Chapter 4

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 in this section.

4.1 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 data into the storage node. As discussed

in Section 2, the whole dataset is partitioned into many data blocks, and the aggregators scatter/gather data between MPI processes iteratively.

Listing 4.2 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 [27]). In the figure, in each iteration, before each collective data write (Line 10), data shuffling will be called to gather data from MPI processes (Line 7).

Listing 4.3 shows the logic of data shuffling in each iteration. Data shuffling is implemented based on an MPI collective communication (particularly MPI_Alltoall) and MPI asynchronized point-to-point communication (MPI_Irecv/MPI_Isend and MPI_Waitall).

Based on the above discussion, we conclude that data shuffling in MPI collective I/O is interleaved with IO operation across iterations. In each iteration, data shuffling must be finished before the aggregator starts to write (or read) data. Assuming the performance is dominated by the slowest aggregator, then the shuffling and IO can be treated as a blocking operation, which simplifies our IO modeling.

In Listings 4.1 and 4.2, we have *ntimes* of iterations and each iteration sends one data piece with the size of "collective buffer".

Listing 4.1: Pseudocode for MPI collective read operations

```
ADIOI_read_and_Exch(...)
{
    ...
```

```
  for (m=0; m<ntimes; m++) {
      ...
      // Contiguous read from storage
      ADIO_ReadContig(...);
      ...
      // Shuffling data between MPI processes
      ADIOI_R_Exchange_data(...);
      ...
  }
  for (m=ntimes; m<max_ntimes; m++) {
      // Nothing to send, but check for write.
      ADIOI_R_Exchange_data(...); basicstyle=\small
  }
  ...
}
```

Listing 4.2: Pseudocode for MPI collective write operations

```
ADIOI_Exch_and_write(...)
{
  ...
  for (m=0; m<ntimes; m++) {
```

```
        ...
    // Shuffling data between MPI processes
        ADIOI_R_Exchange_data(...);
    ...
    // Contiguous write to storage
    ADIO_WriteContig(...);
    ...
  }
  for (m=ntimes; m<max_ntimes; m++) {
    // Nothing to receive, but check for send.
        ADIOI_R_Exchange_data(...);
  }
  ...
}
```

Between each collective I/O iteration, there's no overlap between because each iteration has a barrier in the end. Like Table II shows, every iteration will be blocked until last iteration has been completed.

Listing 4.3: Pseudocode for data shuffle in MPI collective I/O read

```
ADIOI_R_Exchange_data(...)
{
  MPI_Alltoall(...);
```

```
    ...
    for (i=0; i < nprocs; i++) {
        MPI_Irecv(...)
    }
    ...
    for (i=0; i < nprocs; i++) {
        MPI_Isend(...)
    }
    ...
    MPI_Waitall(...)
    ADIOI_Fill_user_buffer(...)
    MPI_Waitall(...)
}
```

4.2 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. Through the timers, we log the performance for each data piece (i.e., data exchanged during one iteration of the data shuffling loop in Listings 4.1 and 4.2).

Take this attribute, we used a timer to record each phase in every iteration. Average

I/O time Tio will indicate device I/O speed, while average shuffle time Ts will indicate time spent in data shuffle. Tio will varies according to device speed, ratio of Ts/Tio will indicate how shuffle time impact in overall collective I/O.

During profiling, we use the same five nodes as Section 3.3. Among the five nodes, four of them run the IOR benchmark and one works as a storage node. For the IOR benchmark, we use 16 processes (4 processes per node), 2 data segment counts, 512 MB block size. Total workload size sum up is 16GB. Transfer size for each collective iteration is 16MB. We use one aggregator per node. Table 4.1 shows our profiling results.

Table 4.1: Profiling results for MPI collective I/O with IOR

Item	**HDD**	**SSD**	**NVM**
I/O time	5938.91s	1002.93	986.15s
Shuffle time	466.21s	499.30s	494.61s
Ratio (shuffle time to collective I/O time)	7.85%	49.93%	50.16%
Average IO time per iteration	170.38ms	28.77ms	28.29ms
Average shuffle time per iteration	13.77ms	14.32ms	14.19ms
IO time variance	0.15	0.05	0.03
Shuffle time variance	0.03	0.02	0.02

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, as we use NVM as the storage device. Note that the shuffle time remains stable across the cases of HDD, SSD, and NVM, because we use the same MPI implementation for MPI collective I/O, same I/O workload, and client configuration, which results in the same communication pattern for the three cases.

4.3 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 4.2.

Table 4.2: Notation for 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 collective buffer size in the collective I/O implementation.
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.

MPI collective I/O ($T_{collective}$) is generally modeled in Equation 4.1. The equation includes the data shuffling time (T_{comm}), I/O operation time (T_{IO}), and other performance cost because of the implementation of MPI collective I/O (T_{other}). T_{comm} and T_{IO} depend on data size and data access patterns of MPI processes. We model them as follows.

$$T_{collective} = T_{comm} + T_{IO} + T_{other} \tag{4.1}$$

Data shuffling time (T_{comm}) is modeled in Equation 4.2. T_{comm} is for one MPI 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 a subset of MPI processes.

In Equation 4.2, 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 total data from one MPI process for doing I/O operation, is communicated. Note that Equation 4.2 is only for modeling collective I/O. However, a part of the total data from one MPI process for doing I/O operation may not be involved in the collective I/O. Instead, they are involved in individual I/O. If a part of the total data is already contiguous and friendly for doing I/O operations individually, the collective I/O may not be applied for that part of the data. To capture the above fact, we introduce a parameter τ. $msg\_size_i \times \tau$ is the data really involved in the collective I/O and communicated between an MPI process and the aggregator. 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 $T\_s + T\_w \times msg\_size_i \times \tau$. T_s represents the communication time unrelated with the message size, such as the communication initialization time; T_w represents the communication time

related with the message size (or more precisely speaking, in proportion to the message size).

$$T_{comm} = \sum_{i=1}^{iter} (T_s + T_w \times msg\_size_i \times \tau) \tag{4.2}$$

I/O operation time (T_{IO}) is modeled in Equation 4.3. The numerator of the equation is the data that has been shuffled and ready for I/O operation, as discussed above in T_{comm}. 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). bdw_{seq} is the bandwidth for doing sequential I/O, because after data shuffling, there is supposed to be sequential data access between the aggregator and storage node.

$$T_{IO} = \frac{\sum_{i=1}^{iter} msg\_size_i \times \tau}{bdw_{seq}} \tag{4.3}$$

T_{other} in Equation 4.1 is the other performance cost besides data shuffling, including memory mapping, variable initialization, system logs, and data checking for data alignment (See Lines 13-15 in Listing 4.1 and Listing 4.2).

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 4.4. $T_{individual}$ 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 4.5. 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.

$$T_{individual} = T_{IO} + T_{other} \tag{4.4}$$

$$T_{IO} = \frac{\sum_{i=1}^{iter} msg\_size_i}{bdw_{ran}} \tag{4.5}$$

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).

bdw_{seq} and bdw_{ran} can be measured by using the IOR benchmark. In particular, we deploy IOR on our test environment with the 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 4.3 summarize the results

in our test platform. One interesting thing 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 larger device bandwidth than HDD, such that the end-to-end bandwidth is limited by networking.

Table 4.3: 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 the IOR benchmark. 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 times and shuffling time. Then, we calculate T_{other} based on Equations 4.1 and 4.4 for collective I/O and individual I/O respectively. In our tests, we find T_{other} is much smaller than I/O operation time and data shuffling time, hence we set it as zero during model verification (Section 4.3).

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

The parameter τ heavily depends on the application data access pattern and MPI imple-

mentation. It is challenging to choose a universal value for all cases. Also, it is challenging to ask the user to quantify their workload characteristics and choose τ. We use an empirical-based approach to decide τ. In particular, we ask the user to qualitatively decide if individual I/O operations are able to be coalesced. If yes, then we set τ as 1, otherwise we set τ as 0.2.

Discussion. Our model has two limitations. First, we do not distinguish intra- and inter-node communication in Equation 4.2 when modeling data shuffling time. In particular, we measure T_s and T_w based on inter-node communication and use them in Equation 4.2, no matter whether data shuffling happens within a node or between nodes. Second, we assume that all aggregators have roughly the same data shuffling time, such that the data shuffling times of all aggregators are overlapped. However, depending on data access patterns of each MPI process, different aggregators working with different MPI processes can have different 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 into nodes 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 helps for modeling accuracy. Hence, we do not assume such knowledge is available in our model.

Model Verification

We verify our model accuracy with IOR on five nodes, four of which are compute nodes and one of which is a storage node, the same as our previous multi-node evaluation. We use four MPI processes per node (16 MPI processes in total). With IOR, we evaluate two types of workloads. One type of workload has random but overlapped data accesses from MPI processes. This is achieved by enable "reorder tasks to random". For this type of workload, τ is set as 1 to predict collective I/O performance. The other type of workload has sequential, non-overlapped data access pattern for each MPI process. For this type of workload, τ is set as 0.2 to predict collective I/O performance. This also indicates that 20% of data for I/O operations are based on the model for collective I/O, while 80% of data for I/O operations are based on the model for individual I/O. For all tests, the IOR parameters, "segment count" is set as 2, "block size" is set as 64MB, and the collective buffer size is 16MB.

Collective I/O time

Before using the model, we need to get the measured values of T_w and T_s. We used a smaller IOR benchmark collectively read and write through multiple nodes (4 computing nodes and 1 storage node as same as experiments above), 2 segment count, 64MB block size, 4 processors. We set collective buffer size (msg_size) to 4MB, 8MB, 16MB, 64MB representatively and gathering all shuffle times ($shuffleTimes$) reported by MPICH. T_w

and T_s could be get as simply through a linear regression.

Table 4.4: Shuffle time phase test

Transfer buffer size in IOR (MB)	2	4	8	16
Average shuffle time (s)	0.072	0.136	0.283	0.547

Iteration time ($iter$) is determined by workload size divide the message size "msg_size" for each collective I/O iteration. To simplify our test, we set IOR to random read back order and pass an "always do collective I/O" hint to MPICH. So, every iteration in our experiment will do collective I/O. Which means collective I/O ratio (τ) in this case is 1.0. Table 4.5 shows all computed value of notations.

Table 4.5 shows the result we collected in our environment.

Table 4.5: Notation values for data shuffle time

T_w	T_s	$iter$	τ
$3.35e-2$	$5.39e-3$	1024	1.0

Through the results above, we could get our computed data shuffle phase time is 138.58s 4.6:

$$\begin{aligned} T_{comm} &= \sum_{i=1}^{iter}(T_s + T_w \times msg\_size_i \times \tau) \\ &= \sum_{i=1}^{1024}(5.39e-3 + 3.35e-2 \times 16 \times 1.0)/4 \\ &= 138.58s \end{aligned} \tag{4.6}$$

Now, the estimated IO time could be computed using Equation 4.3. Table 4.6 shows computed results for three devices.

Table 4.6: Computed IO time for different devices

	HDD	**SSD**	**NVM**
Time (s)	273.20	135.91	133.69

For other time (T_{other}), we continue to use the simple read and write benchmark above. We used individual I/O to manipulate different workload sizes, recording the total times and IO times representatively. Subtracting Total time by IO time could get the other time (T_{other}). The result shows, compared to data shuffle and I/O, T_{other} is small enough to ignore. So, in our verification, we set value of T_{other} as 0.

We have computed the three parts of collective I/O. Sum them up could get the total estimated time. Table 4.7 compares the estimated times with the real run times reported by IOR benchmark.

Table 4.7: Comparison of estimated and measured collective I/O time for the first workload with random data accesses.

Device	**HDD**	**SSD**	**NVM**
Estimated time (s)	411.78	286.21	284.46
Measured time (s)	385.86	277.46	242.54

Comparing the estimate time from our model and the real running time printed from benchmark, we can find that the accuracy of our model is between $88.67\% - 98.92\%$.

Table 4.8: Comparison of estimated and measured individual I/O time for the second workload with sequential data accesses

Device	**HDD**	**SSD**	**NVM**
Estimated time (s)	613.17	160.84	145.88
Measured time (s)	593.04	146.50	146.35

Individual I/O time

Compared to collective I/O, individual I/O crosses out the data shuffle part, and we set T_{other} to 0. The estimated $T_{individual}$ is more likely equal to T_{IO}. Table 4.8 compares the estimate time with real time reported by IOR.

As we discussed in 4.2, the ratio of how much data will be shuffled is determined by certain work I/O pattern. In our experiments, we use a IOR pattern whose data will always trigger data shuffle. However, in real life, not every workload well has this situation.

We calculated the estimated performance of three different devices by given different value of τ to simulate different work situations. Figure 4.1, Figure 4.2 and Figure 4.3 show our prediction results.

From the figures we could predict, HDD will always speed up the performance using collective I/O because the speed-up always beats the overhead of data shuffling. However, most situation in SSD and NVM have reverse results. Which means, once the workload has some interleaves that will trigger collective I/O (τ greater than 10% in our environment), the overhead of data shuffle will always burden the overall performance.

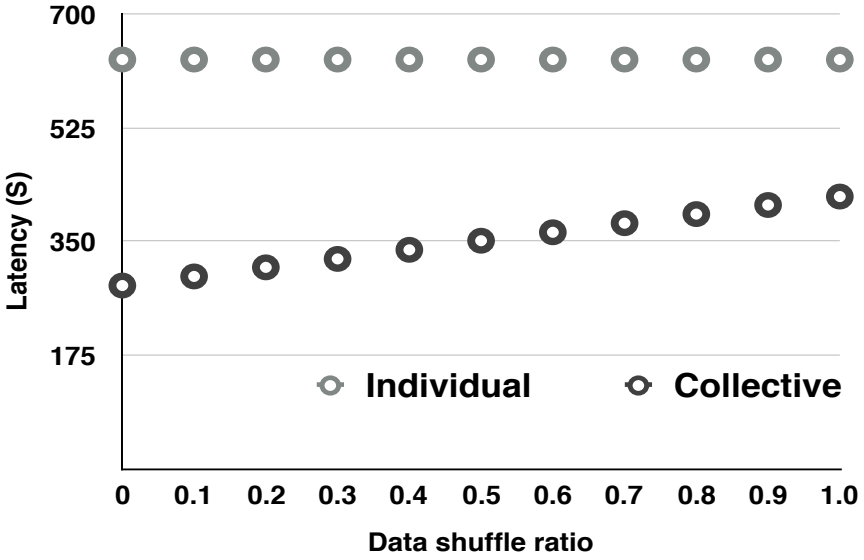

Figure 4.1: Performance prediction of different τ in HDD

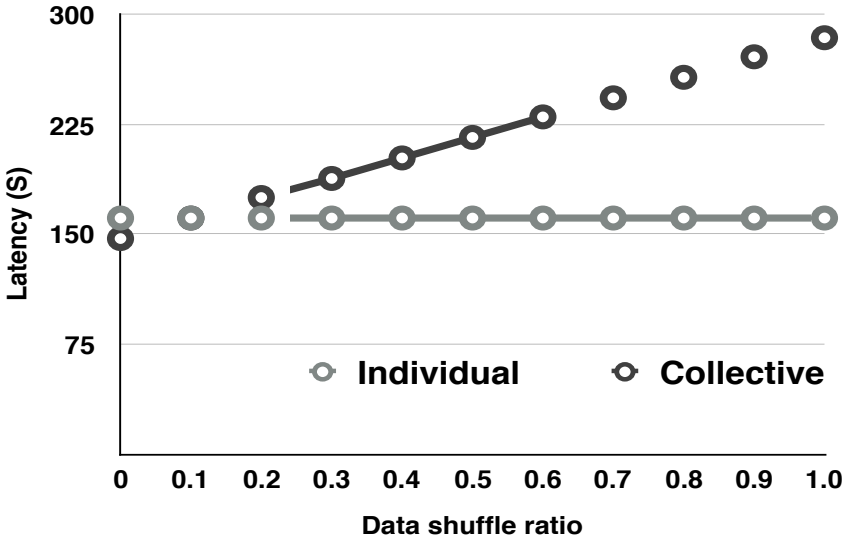

Figure 4.2: Performance prediction of different τ in SSD

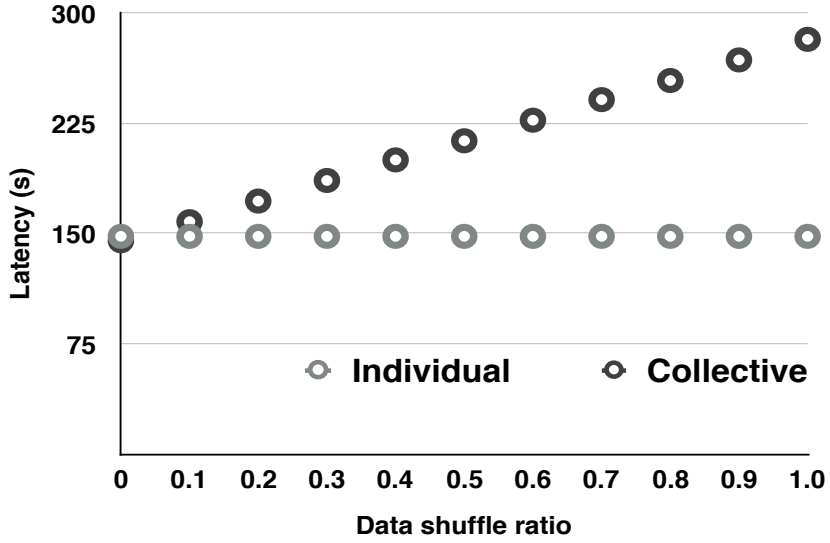

Figure 4.3: Performance prediction of different τ in NVM

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 4.9 and 4.9 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.

Table 4.9: 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 4.10: 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

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.

As a case study, we use our model to study the tradeoff between data shuffling cost (Equation 4.2) 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 mes-

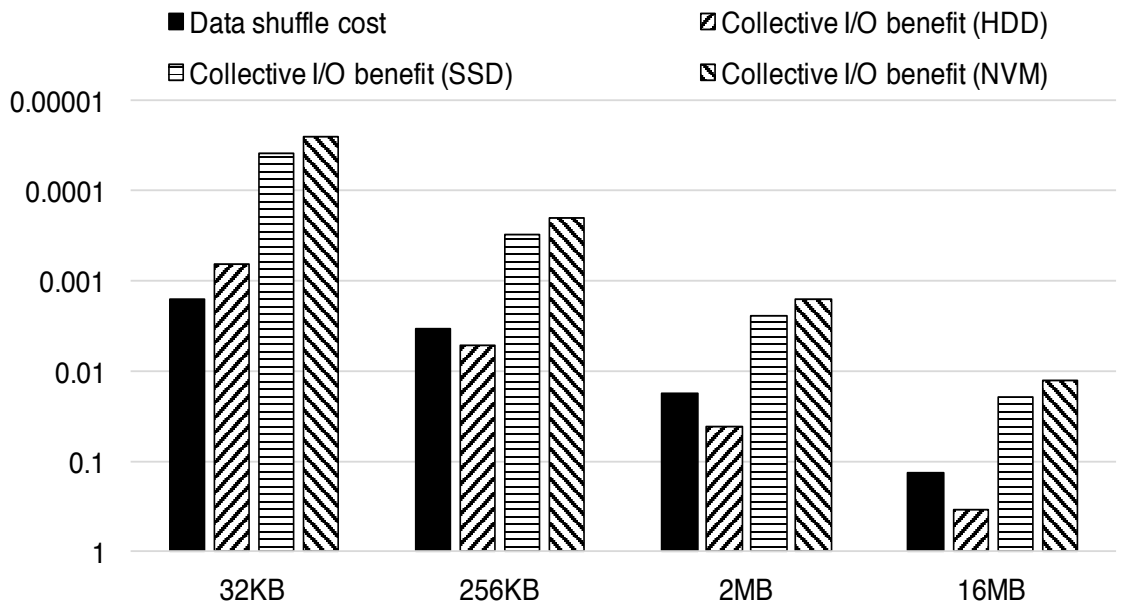

Figure 4.4: Explore the performance tradeoff between data shuffling cost and collective I/O benefit.

sage size. We use bandwidth and communication parameters (i.e., T_w and T_s) measured in our platform for our study. Figure 4.4 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 4.9.

Chapter 5

Related Work

Non-volatile memory. Non-volatile memory (NVM) technology is under quick development and has attracted a large body of research. A comprehensive survey about NVM can be found in a prior study [24]. Here we summarize the most related prior work on NVM.

Prior NVM studies can be roughly classified into several categories. Some earlier studies focus on the architecture-level design issues of NVM [15, 23, 22, 31], such as wear-leveling, read-write disparity issues, etc. Most of these studies are considering NVM as a displacement of DRAM at the architecture level. Another alternative is to consider NVM as a storage device, such as Onyx [1], Moneta [3], and PMBD [7]. The recently announced Intel Optane product [14] 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 [9] uses shadow paging techniques for fast and reliable updates to critical file system metadata structures. SCMFS [30] adopts a scheme similar to page table in mem-

ory management for file management in NVM. PMFS [12] 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 [29] gives a simple programming interface for NVM, such as declaring non-volatile data objects. CDDS [28] attempts to provide consistent and durable data structures. NV-Heaps [8] further gives a simple model with support of transactional semantics; SoftPM [13] 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. Our observations have confirmed that the high-speed NVM could significantly improve HPC application performance, however, the end-to-end effect is also workload dependent and related to a variety of factors in the entire I/O stack, from application, MPI library, OS page cache, file system, to NVM device.

MPI I/O. Aside for hardware, there are also tons of studies related to MPI talks about MPI-IO, collective I/O and software pattern with a focus on improving computing performance, guarantee data safety, and simplify coding strategy.

ROMIO [26] is a widely used implementation of MPI-IO, which is included in MPICH library we were using upon. ROMIO uses two-phase I/O strategy [25]. This technology could reduce latency of access and improve its scalability.

It is still a open topic about determining an optimal number of aggregators for MPI I/O [4]. In this paper, we mainly used one aggregator and used multiple aggregators as

comparison. Because no clear difference shown in experiments about aggregator number, we leave further experimental studies on this issue as our future work.

Several studies have evaluated application efficiency on MPI and tried to improve it. A prior study proposed an optimized buffering system in order to reduce the aggregation cost, as so improving reading and writing data efficiency [25]. Another study found that the nature of collective I/O can have a negative impact on underlying caching algorithm, leading to unnecessary cache misses. An evolution of access pattern proposed address this issue and results in a new collective I/O aware cache management methodology [17]. Another study tried to further reduce the communication costs in collective I/O of multi-core cluster systems with non-exclusive scheduling by regulate the node sequence [5].

Chapter 6

Conclusions

This paper has evaluated the HPC I/O performance based on different storage backend, i.e., HDD, SSD and PMBD. We set experiments by using scientific application benchmarks to test the impact of page cache, performance of POSIX and MPIIO library with NVM. Then we further profiled the collective I/O in ROMIO, which is a major optimization in MPI and HPC I/O. Based on the experimental result, we have developed a model and verified its accuracy. This is a comprehensive study of NVM and its impact to HPC I/O software stack. In conclusion, we found that the I/O performance with NVM is not sensitive to page cache as HDD and SSD are in many cases, so we could shortcut the memory consumption in page cache, improving cost efficiency. We have also confirmed that collective I/O benefit diminishes as the storage becomes faster. So a simplified I/O strategies should be designed and revisited in the era of exascale computing with non-volatile memory.

Bibliography

[1] Ameen Akel et al. "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 2011.

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

[3] Adrian M. Caulfield et al. "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. 2010.

[4] Mohamad Chaarawi and Edgar Gabriel. "Automatically Selecting the Number of Aggregators for Collective I/O Operations". In: *International Conference on Cluster Computing (Cluster)*. 2011.

[5] Kwangho ChaEmail and Seungryoul Maeng. "Reducing communication costs in collective I/O in multi-core cluster systems with non-exclusive scheduling". In: *The Journal of Supercomputing* 61 (3 2012), pp. 966–996.

[6] Feng Chen, Michael P. Mesni, and Scott Hahn. "A Protected Block Device for Persistent Memory". In: *IEEE Symposium on Mass Storage Systems and Technologies (MSST)*. 2014.

[7] Feng Chen, Michael P. Mesnier, and Scott Hahn. "A Protected Block Device for Persistent Memory". In: *Proceedings of the 30th International Conference on Massive Storage Systems and Technology (MSST'14)*. Santa Clara, CA, June 2014.

[8] Joel Coburn et al. "NV-Heaps: Making Persistent Objects Fast and Safe with Next-Generation, Non-Volatile Memory". In: *Proceedings of the 2011 Architectural Support for Programming Languages and Operating Systems (ASLPOS 2011)*. Newport Beach, CA, Mar. 2011.

[9] Jeremy Condit et al. "Better I/O Through Byte-Addressable, Persistent Memory". In: *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP 09)*. Big Sky, MT, Oct. 2009.

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

[11] B. Dieny, R. Sousa G. Prenat, and U. Ebels. "Spin-dependent Phenomena and Their Implementation in Spintronic Devices". In: *Proceedings of 2008 International Symposium onVLSI Technology, Systems and Applications (VLSI-TSA 2008)*. Apr. 2008, pp. 70–71.

[12] Subramanya R Dulloor et al. "System Software for Persistent Memory". In: *Proceedings of the 2014 European Conference on Computer Systems (EuroSys 2014)*. Amsterdam, ST, Netherlands: The ACM, Apr. 2014.

[13] Jorge Guerra et al. "Software Persistent Memory". In: *Proceedings of the 2012 USENIX Annual Technical Conference*. Boston, MA, June 2012.

[14] Intel. https://www.intel.com/OptaneMemory.

[15] B. C. Lee et al. "Architecting Phase Change Memory as a Scalable DRAM Alternative". In: *Proceedings of the 36th International Symposium on Computer Architecture (ISCA 2009)*. 2009.

[16] Benjamin C. Lee et al. "Phase-Change Technology and the Future of Main Memory". In: *IEEE Micro* 30.1 (2010), pp. 143–143.

[17] Yin Lu et al. "Revealing Applications' Access Pattern in Collective I/O for Cache Management". In: *International Conference on Supercomputing (ICS)*. 2014.

[18] James Moorer. "Signal Processing Aspects of Computer Music–A Survey". In: *Computer Music Journal* 1.1 (1977).

[19] *mpiBLAST: Open-Source Parallel BLAST*. http://www.mpiblast.org/.

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

[21] National Energy Research Scientific Computing Center. http://www.nersc.gov/users/computational-systems/cori/nersc-8-procurement/trinity-nersc-8-rfp/nersc-8-trinity-benchmarks/ior/.

[22] 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.

[23] M. K. Qureshi et al. "Enhancing Lifetime and Security of PCM-based Main Memory with Start-gap Wear Leveling". In: *Proceedings of the 42th International Symposium on Microarchitecture (MICRO 2009)*. Dec. 2009.

[24] Kosuke Suzuki and Steven Swanson. *The Non-Volatile Memory Technology Database (NVMDB)*. Tech. rep. CS2015-1011. http://nvmdb.ucsd.edu. Department of Computer Science & Engineering, University of California, San Diego, 2015.

[25] François Tessier et al. "Topology-Aware Data Aggregation for Intensive I/O on Large-Scale Supercomputers". In: *The Workshop on Optimization of Communication in HPC* (2016), pp. 73–81.

[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] 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.

[28] Shivaram Venkataraman et al. "Consistent and Durable Data Structures for Non-volatile Byte-Addressable Memory". In: *Proceedings of the 9th USENIX Conference on File and Storage Technologies (FAST 2011)*. San Jose, CA, Feb. 2011.

[29] Haris Volos, Andres Jaan Tack, and Michael M. Swift. "Mnemosyne: Light Weight Persistent Memory". In: *Proceedings of the 2011 Architectural Support for Programming Languages and Operating Systems (ASLPOS 2011)*. Newport Beach, CA, Mar. 2011.

[30] Xiaojian Wu and A. L. Narasimha Reddy. "SCMFS: A File System for Storage Class Memory". In: *Proceedings of Supercomputing (SC'11)*. Seattle, WA, Nov. 2011.

[31] P. Zhou et al. "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.